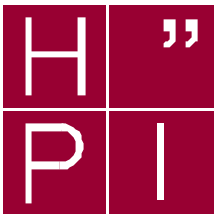
HASSO-PLATTNER-INSTITUT
für Softwaresystemtechnik an der Universität Potsdam



Konzepte der Softwarevisualisierung für komplexe, objektorientierte Softwaresysteme

---

Technische Berichte Nr. 6
des Hasso-Plattner-Instituts
für Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik
an der Universität Potsdam

Nr. 6

Konzepte der Softwarevisualisierung für komplexe, objektorientierte Softwaresysteme

Potsdam 2005

Bibliografische Information der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.ddb.de abrufbar.

Die Reihe *Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam* erscheint aperiodisch.

Herausgeber: Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam

Redaktion: Johannes Bohnet, Jürgen Döllner
Email: Johannes.bohnet, juergen.doellner }@hpi.uni-potsdam.de

Vertrieb: Universitätsverlag Potsdam
Postfach 60 15 53
14415 Potsdam
Fon +49 (0) 331 977 4517
Fax +49 (0) 331 977 4625
e-mail: ubpub@rz.uni-potsdam.de
http://info.ub.uni-potsdam.de/verlag.htm

Druck: allprintmedia gmbH
Blomberger Weg 6a
13437 Berlin
email: info@allprint-media.de

Heft 6 (2005)
ISBN 3-937786-54-6
ISSN 1613-5652

Konzepte der Softwarevisualisierung für komplexe, objektorientierte Softwaresysteme

Vorwort

Johannes Bohnet und Jürgen Döllner

Das Gewinnen von Verständnis über die Struktur und das Verhalten von Softwaresystemen stellt aufgrund der meist immensen Größe und der Evolution der Systeme eine anspruchsvolle, zeit- und kostenintensive Aufgabe dar. Softwarevisualisierung zielt darauf ab, den Prozess der Verständnisgewinnung zu unterstützen, indem sie Zusammenhänge und Einflüsse zwischen Artefakten des Softwaresystems sowie deren Entwicklung offen legt. Die Anwendungsbereiche von Softwarevisualisierung erstrecken sich über den gesamten Entwicklungsprozess eines Softwaresystems.

Ein wesentlicher Schritt für das Verständnis über Softwaresysteme stellt der zunehmend konsequente Einsatz von Software-Modellen dar, wie sie z. B. durch die Unified Modeling Language (UML) definiert sind. Eine Vielzahl von Visualisierungsansätzen vermag es, Aspekte eines Software-Modells zu visualisieren, z. B. in Form automatisiert erstellbarer Diagramme und Graphen.

Die Visualisierungskonzepte, die im Kontext von Software-Modellierung verfügbar sind, setzen die Existenz eines expliziten Modells voraus, das meist vor der System-Implementierung konzipiert und dann fortgeführt wurde. Im Fall eines „gewachsenen" Softwaresystems, das aus einer Vielzahl von Subsystemen und Modulen unterschiedlicher Herkunft, Programmiersprachen und Entwickler aufgebaut ist, kann nahezu ausgeschlossen werden, dass ein aktuelles, vollständiges und homogenes Modell vorliegt.

Im Mittelpunkt der hier vorgelegt Arbeit stehen Konzepte der Softwarevisualisierung, die die Analyse und Exploration komplexer Softwaresysteme auf der Grundlage ihrer Implementierung ermöglichen. Sie nehmen im allgemeinen nicht Bezug auf ein explizit gegebenes Modell von Aspekten der statischen und dynamischen Architektur. Wir sprechen in diesem Zusammenhang von *implementierungsbasierter Softwarevisualisierung*.

Der im Rahmen eines Seminars im Wintersemester 2004/2005 entstandene technische Report gibt einen Überblick über aktuelle Konzepte dieser Form von Softwarevisualisierung komplexer, objektorientierter Softwaresysteme. Die einzelnen Seminarbeiträge zeigen Anwendungsbereiche und identifizieren Stärken und Schwächen ausgewählter Ansätze.

Der Report ist wie folgt gegliedert: Ein einleitender Teil klärt grundlegende Begriffe und Zusammenhänge in der Softwarevisualisierung. Nachfolgend werden in sieben Seminarbeiträgen unterschiedliche Konzepte der Softwarevisualisierung untersucht. Die Seminarbeiträge gehen insbesondere auf die jeweiligen Anwendungsbereiche der Konzepte, den Mehrwert gegenüber anderen Techniken zur Verständnisgewinnung, die Informationsgewinnung, die technischen und konzeptionellen Grenzen und die Automatisierbarkeit ein. Unter anderen werden die folgenden Themenkomplexe angesprochen:

- Verwendung von Metriken in der Softwarevisualisierung
- Quellcodezeilenbasierte Softwarevisualisierung
- Visualisierung der Softwareevolution
- Ausgestaltung des Darstellungsraums in der Softwarevisualisierung
- Exploration von Informationsräumen

Ein abschließender Teil stellt die Ergebnisse der Untersuchungen zusammen. Es werden insbesondere notwendige Kriterien für die Akzeptanz von Softwarevisualisierungs-Werkzeugen identifiziert. In einem Ausblick werden Anwendungsbereiche in der Softwarevisualisierung ausgewiesen, bei denen zukünftige Forschungsaktivitäten erfolgversprechend scheinen.

24. Mai 2005 Die Herausgeber

Konzepte der Softwarevisualisierung für komplexe, objektorientierte Softwaresysteme

Inhaltsverzeichnis

1. Grundlagen der Softwarevisualisierung 1
   Johannes Bohnet und Jürgen Döllner

2. Visualisierung und Exploration von Softwaresystemen mit dem Werkzeug SHriMP/Creole 9
   Alexander Gierak

3. Annex: SHriMP/Creole in der Anwendung 19
   Nebojsa Lazic

4. Metrikbasierte Softwarevisualisierung mit dem Reverse-Engineering-Werkzeug CodeCrawler 27
   Daniel Brinkmann

5. Annex: CodeCrawler in der Anwendung 39
   Benjamin Hagedorn

6. Quellcodezeilenbasierte Softwarevisualisierung 47
   Nebojsa Lazic

7. Landschafts- und Stadtmetaphern zur Softwarevisualisierung 59
   Benjamin Hagedorn

8. Visualisierung von Softwareevolution 73
   Michael Schöbel

9. Ergebnisse und Ausblick 83
   Johannes Bohnet

   Literaturverzeichnis 91

   Autorenverzeichnis 97

KONZEPTE DER SOFTWAREVISUALISIERUNG FÜR KOMPLEXE, OBJEKTORIENTIERTE SOFTWARESYSTEME

Kapitel 1

Grundlagen der Softwarevisualisierung

Johannes Bohnet und Jürgen Döllner

Kapitel 1: Grundlagen der Softwarevisualisierung

Johannes Bohnet und Jürgen Döllner

Zusammenfassung. Das Gewinnen von Verständnis über die Struktur und das Verhalten von Softwaresystemen stellt aufgrund der meist immensen Systemgröße und der Evolution des Systems eine anspruchsvolle, zeit- und kostenintensive Aufgabe in der Softwareentwicklung und Softwarepflege dar. Softwarevisualisierung kann den Prozess der Verständnisgewinnung unterstützen, indem sie Zusammenhänge, Abhängigkeiten und Einflüsse zwischen Artefakten des Softwaresystems offen legt. Softwarevisualisierung auf der Grundlage des implementierten Systems findet Einsatz während des gesamten Entwicklungsprozesses und ist unabhängig von der Anwendungsdomäne einsetzbar. Die zu visualisierenden implementierungsbasierten Informationen können dabei aus der Quellcodebeschreibung des Softwaresystems extrahiert, während des Programmablaufs erzeugt oder über die Betrachtung der zeitlichen Entwicklung des Systems generiert werden. Optional können zusätzlich Informationen über weitere an der Softwareentwicklung beteiligten Aspekte in die Visualisierung einfließen.

1.1 Motivation

Heutige Softwaresysteme gehören mit zu den komplexesten, von Menschen erstellten Systemen. Das Verständnis von Strukturen, Abhängigkeiten, Einflüsse und Verhalten der Systeme und ihrer Komponenten steht im Mittelpunkt von Verfahren der Softwarevisualisierung. Sie verfolgen darüber hinaus das Ziel, Einsicht in die mit ihnen verbundenen Entwicklungs- und Managementprozesse ermöglichen, denen ein langfristig genutztes und fortgeführtes komplexes Softwaresystem unterliegt.

Mit dem Begriff *Komplexität eines Softwaresystems* beziehen wir uns dabei zum einen auf die Größe der Implementierung des Systems, die bei den meisten Legacy-Systemen mehrere Millionen Lines-of-Code umfasst. Zum anderen bezieht sich die Komplexität auf die Vielzahl der beteiligten Entwickler, die über Jahre in unterschiedlichen Rollen und in unterschiedlichen Entwicklungsphase das System aufbauen und fortführen. Weiter bezieht sich die Komplexität auf die über Jahre andauernde Weiterentwicklung und Überarbeitung existierender Systemkomponenten, in deren Folge sich u. a. konzeptionelle Dissonanzen, Implementierungsredundanzen, Architektur-Asymmetrien und stark variierenden Lösungsqualitäten niederschlagen. Allgemein dient Komplexität bei Softwaresystemen als Maß für den Schwierigkeitsgrad, mit dem das Softwaresystem, seine Bestandteile und die mit ihm verbundenen Prozesse seiner Entwicklung und Nutzung verstehbar, nachvollziehbar und überprüfbar sind.

Mit der Komplexität eines Softwaresystems wächst der Aufwand, der insbesondere mittel- und langfristig notwendig ist, um das System zu warten und fortzuentwickeln. Wartung und Pflege repräsentieren den Großteil der insgesamt anfallenden Herstellungskosten eines Softwaresystems [95], so dass Verfahren und Strategien zur Beherrschung der Komplexität von zentralem Interesse sind, die Herstellungskosten zu senken und die Qualität des Softwaresystems zu verbessern.

1.1.1 Softwaremodelle

Ein wesentlicher Schritt zur Beherrschung der Komplexität von Softwaresystemen stellt der zunehmend konsequente Einsatz von Softwaremodellen dar, wie sie z. B. durch die Unified Modeling Language (UML), definiert sind. „UML helps you specify, visualize, and document models of software systems, including their structure and design […]" [75]. Ein solches Modell existiert nebenläufig zu der Implementierung des Softwaresystems. Um die Konsistenz zwischen Modell und Implementierung zu gewährleisten, muss daher nach einer Modifikation des Modells der Quellcode entsprechend der Modelländerung angepasst werden. Umgekehrt muss nach einer Modifikation des Quellcodes geprüft werden, ob die Implementierung noch immer dem Modell entspricht, wobei gegebenenfalls eine Anpassung des Modells notwendig ist. Diese Anpassung ist teilweise, z. B. für Klassendiagramme, automatisiert möglich. Aktivitätsdiagramme hingegen lassen sich nur durch eine manuelle Modellanpassung exakt fortführen. Eine Gewährleistung, dass das explizite Modell die tatsächliche Implementierung des Softwaresystems beschreibt, ist nur durch ein konsequentes Einhalten des wechselseitigen Zyklus von Quellcodeanpassung und Modellanpassung gegeben.

In der Praxis gestaltet sich die Einhaltung dieses Zyklus schwierig, da insbesondere bei der

Verwendung von agilen Methoden in der Softwareentwicklung [1] wie z. B. dem eXtreme Programming [32] schnelle Iterationszyklen zur Auslieferung funktionierender Software erwünscht sind, um möglichst schnell die Funktionalität der Software auf Kundenwünsche anpassen zu können. Die Angleichung eines nebenläufig zur Implementierung existierenden Modells steht im Hintergrund. Ferner ergeben sich Dissonanzen zwischen expliziten Systemmodellen und der Implementierung im weiteren Verlauf des Lebenszyklus des Softwareproduktes während seiner Weiterentwicklung und der Überarbeitung existierender Systemkomponenten.

Die Gewährleistung, dass insbesondere mit zunehmenden Alter eines Softwaresystems ein aktuelles und vollständiges Systemmodell vorliegt, das widerspruchsfrei zu der Implementierung ist, kann also nicht gegeben werden. Eine Ausnahme besteht bei der Verwendung von Modell-getriebenen Softwareentwicklungs-Ansätzen (Model Driven Architecture MDA) [76], bei denen der Quellcode automatisch aus einem vollständigen Modell generiert wird. Die Entwicklung der Software findet ausschließlich auf der Abstraktionsebene des Modells statt. In der Praxis werden MDA-Verfahren in Großprojekten schon verwendet, allerdings ist die Quellcodegenerierung nicht vollständig, so dass große Teile der Implementierung weiterhin explizit von Entwicklern geschrieben werden müssen (siehe z. B. [89] oder [59]).

Ein verlässliches Verständnis von der Struktur, dem Verhalten, von Abhängigkeiten und Einflüssen eines Softwaresystems und seiner Komponenten kann daher nur auf Basis seiner Implementierung gewonnen werden.

1.1.2 Softwarevisualisierung

Softwarevisualisierung stellt Mittel bereit, die die Verständnisgewinnung über Softwaresysteme unterstützen. Sie umfasst „[...] the development and evaluation of methods for graphically representing different aspects of software, including its structure, its abstract and concrete execution, and its evolution" [84].

In der Softwarevisualisierung können drei Aufgabenbereiche unterschieden werden, in denen die Verständnisgewinnung eine besondere Rolle spielt:

1. Konzeption und Entwicklung neuer Softwaresystemen;
2. Wartung, Erweiterung und Wiederverwendung existierender Softwaresysteme;
3. Management und Monitoring des Software-Entwicklungsprozesses.

Bei der Konzeption und Entwicklung neuer Softwaresysteme finden im Kontext der Objektorientierung Softwarevisualisierungstechniken Einsatz, die auf der Visualisierung von Systemmodellen beruhen. Die Modelle, die in Diagrammform statische Aspekte der Softwarearchitektur, dynamische Aspekte der Systemausführung und physische Aspekte der Systemoperationalisierung umfassen, werden durch die Entwickler erstellt und mit der Implementierung umgesetzt. Sie bleiben nach der Implementierung nur teilweise an diese gekoppelt. Für das Forward-Engineering stehen mit der Unified Modeling Language leistungsstarke, klassische Modellierungs- und Visualisierungsansätze bereit.

Bei der Wartung, Erweiterung und Wiederverwendung hingegen erweisen sich die Methoden aus dem Forward-Engineering als nur bedingt einsetzbar, da im allgemeinen keine explizit entwickelten oder aktuellen Systemmodelle bereitstehen. Das Reverse-Engineering stellt jedoch den Hauptfall des Software-Engineerings in der Zukunft dar [37]. Die damit verbundenen Aufgaben umfassen die funktionale Erweiterung des Systems, die Fehlerbehebung und die Optimierung. Zur Durchführung wird auf unterschiedliche Methoden, wie z. B. das Refactoring, zurückgegriffen. Softwarevisualisierungstechniken, die automatisiert Informationen über Struktur, Abhängigkeiten und Dynamik generieren und vermitteln können, helfen diese Aufgaben zeit- und kosteneffektiv zu bewältigen. Zudem lassen sich mit geeigneten Visualisierungstechniken Systemteile extrahieren, die Hinweise auf ein besonderes Verhalten aufweisen. So können z. B. fehleranfällige oder funktionsüberladende Systemteile leichter identifiziert werden.

Durch die Verknüpfung von Informationen, die aus dem Quellcode gewonnen werden können, mit Informationen, die Aufschluss über die zeitliche Entwicklung des Softwaresystems durch Entwickler geben, lassen sich Visualisierungstechniken für Management- oder Controlling-Aufgaben entwickeln. Hierdurch kann ein Monitoring des Softwareentwicklungsprozesses erfolgen und Entscheidungsunterstützung gegeben werden.

1.2 Grundlagen

In der Literatur wird bei der Softwarevisualisierung zwischen Programm- und Algorithmus-Visualisierung unterschieden [81]. Im folgenden werden unter dem Begriff *Softwarevisualisierung* Visualisierungstechniken aus beiden Bereichen verstanden, die auf die Verständnisgewinnung über die Struktur, das Verhalten und die Evolution von komplexen Softwaresystemen zielen. Wir

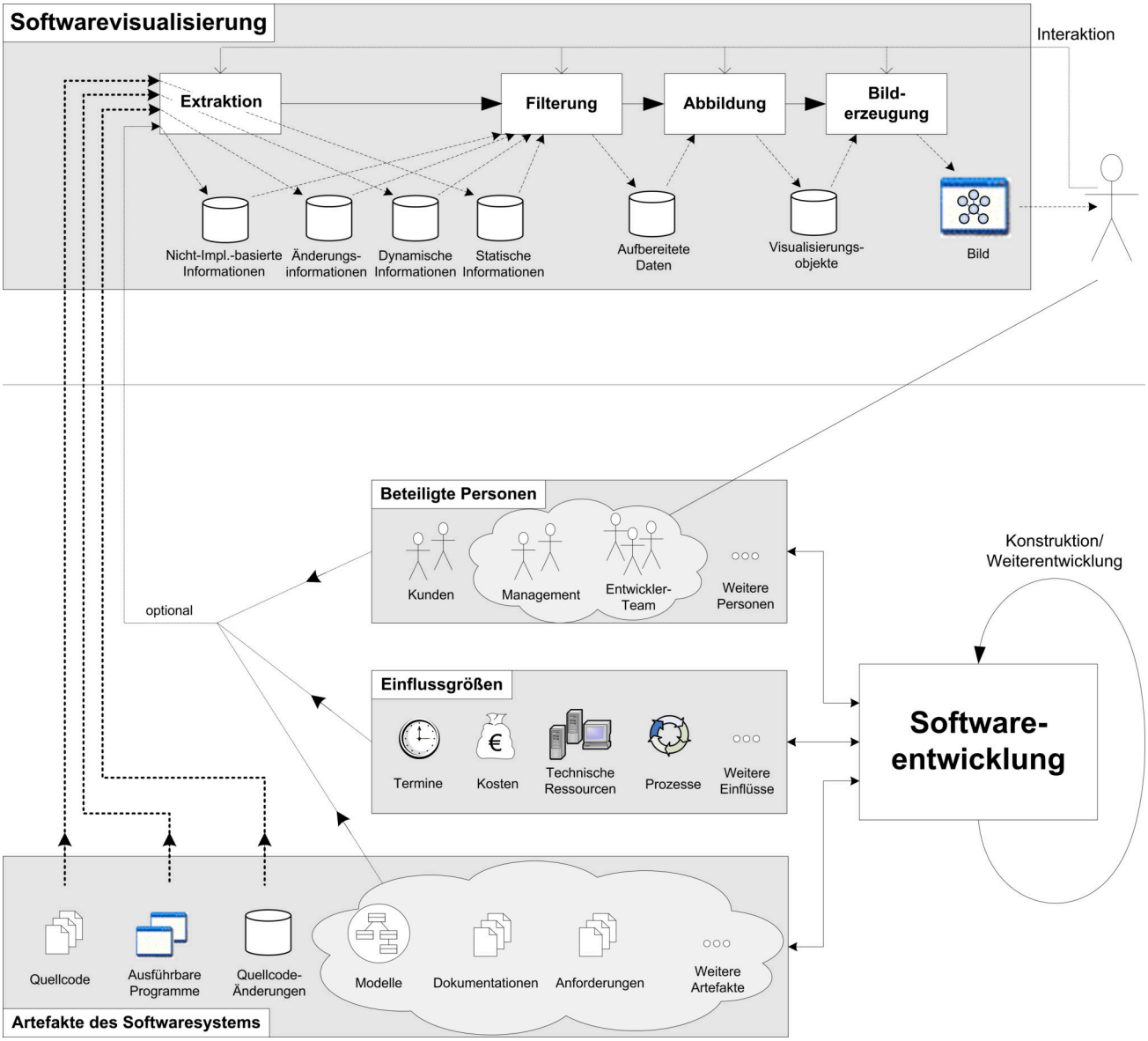


Abbildung 1: Implementierungsbasierte Softwarevisualisierung.

schränken den Begriff der Softwarevisualisierung weiter ein, indem wir Visualisierungen betrachten, die auf Basis der Implementierung eines Softwaresystemen erstellt werden.

1.2.1 Softwarevisualisierung in der Softwareentwicklung

Abbildung 1 skizziert, wie die Softwarevisualisierung in den Prozess der Softwareentwicklung eingebettet ist:

Drei Gruppen von Aspekten spielen bei der Softwareentwicklung eine Rolle:

1) Beteiligte Personengruppen;
2) Einflussgrößen;
3) Artefakte des Softwaresystems.

Die wichtigsten Personengruppen, die an der Konstruktion und Weiterentwicklung von Softwaresystemen beteiligt sind, sind das Entwicklerteam, das Management und die Kunden. Termine, Kosten, technische Ressourcen und Prozesse stellen Einflussgrößen während der Softwareentwicklung dar. Das Softwaresystem selbst besteht aus einer Menge von Artefakten, die während der Entwicklung erzeugt und modifiziert werden:

- Quellcode;
- Ausführbare Programme;
- Informationen über Quellcodeänderungen;
- Explizite Modelle über die Struktur oder das Verhalten des Systems (z. B. in Form eines UML-Modells);
- Dokumentationen;
- Anforderungen;
- Weitere Artefakte.

Softwarevisualisierung zielt auf die Unterstützung der an der Softwareentwicklung beteiligten Personengruppen, insbesondere Entwickler und Mana-

ger, bei der Verständnisgewinnung über das Softwaresystem. Für die Visualisierung sind als Datenquellen implementierungsbezogene Artefakte von besonderem Interesse, da diese untereinander konsistenten Artefakte die Struktur, das Verhalten und die Evolution des Softwaresystems vollständig definieren:

- Quellcode;
- Ausführbare Programme;
- Informationen über Quellcodeänderungen.

Optional können neben den implementierungsbezogenen Artefakten sowohl Informationen über weitere Softwareartefakte als auch Informationen über Einflussgrößen oder beteiligte Personengruppen in die Visualisierung eingehen.

Explizite Systemmodelle oder Systemdokumentationen sind möglicherweise veraltet und weisen Widersprüche zu den implementierungsbezogenen Artefakten auf. Sie stellen daher keine verlässliche Beschreibung des Systems dar. Die Überprüfung dieser Artefakte auf ihre Aktualität hin stellt daher eine interessante Anwendung für die Softwarevisualisierung dar.

Das Verknüpfen von Informationen über implementierungsbezogenen Softwareartefakte mit weiteren Aspekten des Entwicklungsprozesses ist für die Unterstützung von Management-Aufgaben sinnvoll. Visualisierungen dieser Zusammenhänge bieten Entscheidungshilfen für das Management und fördern eine Optimierung des Entwicklungsprozesses, indem sie mögliche Schwachpunkte des Prozesses ersichtlich machen. Problematisch hierbei ist jedoch, dass sich diese zusätzlichen Informationen im Allgemeinen nicht automatisiert gewinnen lassen. Eine Ausnahme bilden Informationen über die Softwareentwicklung, wenn ein Versionsverwaltungssystem genutzt wird. Neben den Änderungen am Quellcode werden Angaben über die für die Änderungen verantwortlichen Entwickler gespeichert. Diese zusätzlichen Informationen können zusammen mit den Quellcodeänderungen vollständig automatisiert vom Visualisierungssystem erfasst werden.

1.2.2 Softwarevisualisierungs-Pipeline

In Abbildung 1 ist ein Modell einer Softwarevisualisierungs-Pipeline dargestellt. Es orientiert sich an dem allgemeinen Modell der Visualisierungspipeline [91]. Sie beinhaltet vier sequentielle Schritte:

1) Extraktion;
2) Filterung;
3) Abbildung;
4) Bilderzeugung.

Nach dem Extraktionsschritt stehen für die Visualisierungs-Pipeline unterschiedliche Typen von Informationen über das Softwaresystem zur Verfügung:

- Informationen, die aus dem Quellcode, extrahiert werden (*statische Informationen*);
- Informationen, die während der Programmausführung erzeugt werden (*dynamische Informationen*);
- Informationen, die über die zeitliche Entwicklung des Softwaresystems im Softwareentwicklungsprozess Aufschluss geben (*Änderungsinformationen*);
- Informationen über weitere Softwareartefakte, Einflussgrößen bei der Entwicklung oder über an der Entwicklung beteiligte Personen (*Nichtimplementierungsbasierte Informationen*).

Im Filterungsschritt werden diese Rohdaten durch Reduktion und Verknüpfung aufbereitet und in einem weiteren Schritt, der Abbildung, Visualisierungsobjekten zugeordnet. Das heißt, es wird ein Modell erzeugt, in dem geometrische Primitive nach Konstruktionsregeln im Darstellungsraum angeordnet werden. Die Primitive können mit Attributen, wie z. B. einer Farbe, versehen sein. In der Bilderzeugung wird in Abhängigkeit von Parametern wie z. B. der Betrachterposition aus dem geometrischen Modell ein Bild generiert, das von den Nutzern betrachtet wird.

Für ein interaktives Explorieren des dargestellten Informationsraums müssen Methoden zur Navigation bereitgestellt werden. Dies geschieht, indem den Nutzern ein Eingreifen in die Bilderzeugung erlaubt wird. Idealerweise können zudem die Extraktion, die Filterung und die Abbildung von den Nutzern konfiguriert werden.

1.3 Softwarevisualisierungs-Werkzeuge

Für die Akzeptanz eines Softwarevisualisierung-Werkzeuges ist es notwendig, dass der Prozess der Visualisierungserzeugung weitgehend automatisiert abläuft. Dies gilt zum einen für die Informationsgewinnung und zum anderen für die Anordnung der grafischen Elemente in der Visualisierung (Layout).

Für die sinnvolle Verwendung eines Softwarevisualisierungs-Werkzeugs ist zudem der Aspekt der Skalierbarkeit zu berücksichtigen. Informationsvisualisierung erreicht unter anderem dadurch einen Mehrwert, dass die menschliche Fähigkeit der Mustererkennung unterstützt wird. Dass heißt,

dass Softwarevisualisierung vor allem bei großen Informationsvolumen einen signifikanten Vorteil gegenüber z. B. der direkten Quellcodebetrachtung bietet. Der Erzeugungs- bzw. Extraktionsschritt von Informationen über ein Softwaresystem durch ein Softwarevisualisierungs-Werkzeug sollte daher große Datenmengen verarbeiten können. Ebenso sollte das Visualisierungsprinzip für große Datenmengen ausgelegt sein. Dabei sollte eine interaktive Exploration der Informationen möglich sein.

Ein weitere Aspekt, der für die Akzeptanz eines Softwarevisualisierungs-Werkzeug eine Rolle spielt, ist die Einbettung desselben in eine Entwicklungsumgebung. Die parallele Verwendung des *stand-alone* Visualisierungs-Werkzeugs bei der Softwareentwicklung bedeutet einen Mehraufwand für die Nutzer und erhöht somit die Hemmschwelle für die Nutzung.

1.4 Implementierungsbasierte Softwarevisualisierung

Ziel des Reports *Konzepte der Softwarevisualisierung für komplexe, objektorientierte Softwaresysteme* ist es, einen Überblick über aktuelle Konzepte im Bereich der Softwarevisualisierung zu geben. Es wird dabei u.a. auf folgende Fragestellungen eingegangen:

- Für welchen Anwendungsbereich ist welches Visualisierungskonzept sinnvoll?
- Wann ergibt sich durch die Visualisierung ein Mehrwert gegenüber anderen Techniken zur Verständnisgewinnung?
- Welche Informationen können visualisiert werden, und wie werden diese erzeugt?
- Wo liegen konzeptionelle bzw. technische Grenzen des jeweiligen Visualisierungsansatzes?
- Wie weit ist der Visualisierungsprozess automatisierbar?

In fünf Arbeitspaketen werden unterschiedliche Aspekte der Softwarevisualisierung untersucht:

- Der Fokus bei der Analyse des Softwarevisualisierungs-Werkzeugs SHriMP/Creole [97] liegt auf der Untersuchung graphbasierter Visualisierungstechniken und speziell der Exploration von hierarchischen Informationsräumen. Die bei SHriMP/Creole visualisierten Informationen sind statische Elemente des UML-Modells.
- Bei der Untersuchung des Softwarevisualisierungs-Werkzeugs CodeCrawler [15] wird auf die Bedeutung von Softwaremetriken bei der Softwarevisualisierung eingegangen.
- Codezeilenbasierte Visualisierungstechniken gehören mit zu den ältesten verwendeten Techniken im Bereich der Softwarevisualisierung und werden noch immer in aktuellen Konzepten genutzt. Die Gründe hierfür werden untersucht. Dabei werden diese Techniken, mit denen Softwaresysteme auf einer niedrigen Abstraktionsebene abgebildet werden, analysiert.
- Die Beschränkung der Informationsdichte bei 2-dimensionalen Darstellungen lässt sich durch eine Erweiterung des Darstellungsraums auf drei Dimensionen umgehen. Bei der Untersuchung von Landschafts- und Stadtmetaphern als Basis für Visualisierungstechniken im 2½-dimensionalen Darstellungsraum wird die Bedeutung der Dimension des Darstellungsraums erörtert und Konzepte zur Softwarevisualisierung basierend auf den genannten Metaphern analysiert.
- Die Visualisierung der Evolution eines Softwaresystems kann z. B. für Management-Aufgaben sinnvoll sein. In der Betrachtung entsprechender Konzepte wird zum einen auf die Entwicklung des Quellcodes und damit der Funktionalität der Softwareartefakte im Laufe der Zeit eingegangen. Zum anderen wird analysiert, wie die arbeitsteilige Entwicklung von Softwaresystemen durch Visualisierung unterstützt werden kann.

Kapitel 2

Visualisierung und Exploration von Softwaresystemen mit dem Werkzeug SHriMP/Creole

ALEXANDER GIERAK

Kapitel 2: Visualisierung und Exploration von Softwaresystemen mit dem Werkzeug SHriMP/Creole

Alexander Gierak

Zusammenfassung. Die Softwarevisualisierung erforscht die wissenschaftlichen Grundlagen zur Darstellung von Softwaresystemen und setzt die gefundenen Ergebnisse und Konzepte mittels geeigneter Werkzeuge um. Dieser Artikel stellt die Technologie SHriMP – (Simple Hierarchical Multi-Perspective Views) vor und gibt einen Überblick über die benutzten Techniken und Konzepte. Am Beispiel des OpenCMS Softwaresystems wird eine typische Analyse mit Creole, das auf SHriMP basiert, gezeigt. Anhand dieser Analyse werden die Stärken und Schwächen von Creole herausgearbeitet und eine umfassende Bewertung abgegeben.

2.1 Einführung

Software wird für die Maschine ausreichend durch den Quelltext beschrieben, so dass er von Compilern oder Interpretern übersetzt bzw. ausgeführt werden kann. Für den Menschen allerdings, speziell für Softwareentwickler, ist der Quelltext anderer Entwickler dagegen sehr schwer verständlich. Es ist großer kognitiver Aufwand notwendig, um aus vielen Quelltextdateien die Softwarestrukturen zu erkennen und ein Systemverständnis zu erlangen. Dieses Verständnis ist allerdings Voraussetzung für die Durchführung von Programmieraufgaben wie Weiterentwicklung und Wartung von Softwaresystemen.

Die Softwarevisualisierung erforscht deshalb Konzepte und Technologien, um dem Softwareentwickler die Strukturen und Eigenschaften, die im Quelltext verborgen sind, verständlich zu visualisieren. Das heißt, die Information sollen so präsentiert werden, dass sie mit weniger kognitivem Aufwand und in kürzerem Zeitraum durch den Softwareentwickler erfasst werden können.

Die Simple Hierarchical Multi-Perspective Views – *SHriMP* sind ein Ansatz zur Visualisierung von komplexen Softwaresystemen. SHriMP wurde an der Universität von Victoria, Victoria Kanada durch die CHiSEL (Computer Human Interaction & Software Engineering Lab) Group unter Leitung von Margaret-Anne Storey entwickelt. Die dabei verwendeten Technologien und Konzepte werden im Kapitel 2 ausführlich beschrieben.

In Abschnitt 3 wird Creole vorgestellt, ein Visualisierungswerkzeug, das auf SHriMP basiert und als Plug-in für die Eclipse Plattform [27] realisiert wurde.

Anschließend wird in Abschnitt 4 die Analyse des Open Source Content Management Systems OpenCMS [77] mit Creole und die dabei erzielten Ergebnisse beschrieben. Es wird die Umsetzung der Technologien und Konzepte von SHriMP in Creole anhand der Analyseergebnisse ausführlich beschrieben und bewertet.

Im 5. Abschnitt erfolgt eine Einordnung und Kategorisierung von Creole, um einen Vergleich mit anderen Visualisierungswerkzeugen zu ermöglichen. Im letzten Kapitel wird ein Fazit gezogen und ein Ausblick auf mögliche Weiterentwicklungen gegeben.

2.2 Konzepte und Techniken

Dieser Abschnitt stellt SHriMP vor. Das Hauptaugenmerk liegt dabei auf den Elementen der Visualisierung. Die Art der Informationen, die visualisiert werden sollen, wird in diesem Abschnitt zunächst vernachlässigt.

2.2.1 Verschachtelte Graph

Die zentrale Idee von SHriMP ist der verschachtelte Graph [97] Dabei werden *Knoten* und *Kanten* dargestellt, die ihrerseits wieder andere Knoten bzw. Kanten enthalten können. Knoten repräsentieren *Entitäten* aus dem zu visualisierendem Informationsraum, Kanten die *Beziehungen* zwischen diesen Entitäten. In dem verschachtelten Graphen entsteht somit eine Hierarchie von Knoten und Kanten. In Abbildung 2 ist ein solcher, nur aus Knoten bestehender Graph, abgebildet.

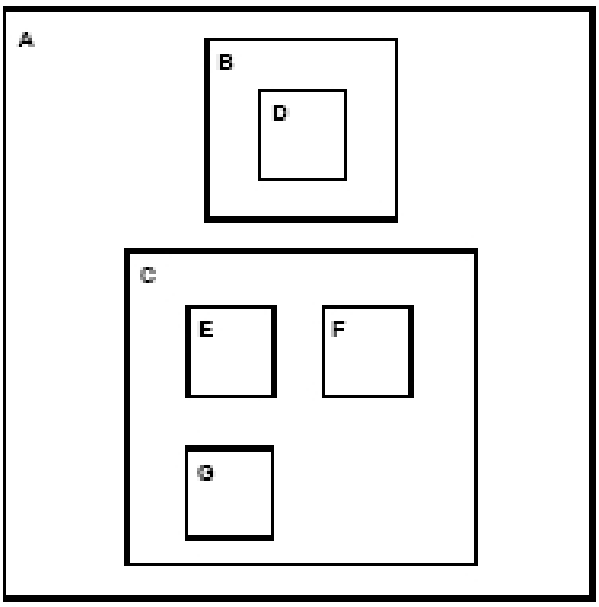


Abbildung 2 Verschachtelter Graph [97]

Es ist leicht zu erkennen, dass Knoten A die Knoten B und C enthält, die wiederum D bzw. E, F und G enthalten. Der Knoten A stellt somit die oberste bzw. erste Ebene der Hierarchie dar. B und C liegen in der zweiten Hierarchieebene. Der Vorteil dieser Struktur besteht darin, dass bei großer Anzahl von Ebenen und Knoten, die Sichtbarkeit bis zu einer bestimmten Ebene eingeschränkt werden kann oder nur einzelne Knoten ihre Kindknoten anzeigen. Dadurch wird die Übersichtlichkeit stark erhöht.

Beziehungen zwischen den Knoten werden durch Kanten ausgedrückt. In der gleichen Hierarchieebene werden die Knoten direkt verbunden. Auf höherer Ebene kann eine zusammengesetzte Kante mehrere Beziehungen auf tieferer Ebene darstellen. Die Breite der Kante gibt dabei die Anzahl der ursprünglichen Kanten an. (Siehe Abbildung 3)

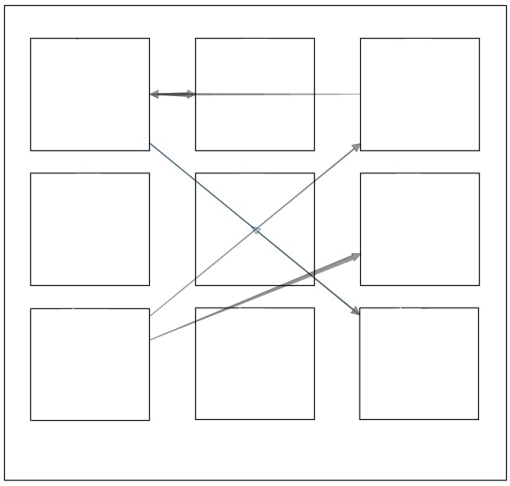


Abbildung 3 Zusammengesetzte Kanten

2.2.2 Geometrische Formen und Farben

Den Knoten und Kanten können bei SHriMP verschiedene geometrische Formen und Farben zugewiesen werden. Es ist somit möglich, unterschiedliche Arten von Knoten bzw. Kanten durch unterschiedliche Formen zu visualisieren. Durch die Zuweisung verschiedener Farben kann diese Unterscheidung noch verstärkt werden oder zusätzliche *Eigenschaften* aus dem Informationsraum dargestellt werden. Die Farbpalette umfasst die möglichen 16,7 Millionen RGB-Farben. Die Knotenformen sind in z.B. in Abbildung 4 erkennbar. Die Kantenformen sind gerade oder geschwungene Pfeile, die aus durchgezogenen bzw. gestrichelten Linien bestehen. (Siehe Abbildung 7)

2.2.3 Layouts

Die Anordnung der Kindknoten innerhalb eines Knotens wird bei SHriMP als Layout bezeichnet. Es stehen mehrere Layoutarten zu Verfügung, um es dem Benutzer zu ermöglichen, für ihn interessante Eigenschaften durch ein geeignetes Layout zu visualisieren.

Das Standardlayout von SHriMP ist das *Gridlayout*. Dabei werden die Knoten anhand eines unsichtbaren Gitters ausgerichtet und haben alle die gleiche Größe. Die Reihenfolge kann dabei eine Eigenschaft der Knoten darstellen, z.B. können die Knoten nach Namen in alphabetischer Reihenfolge sortiert werden. Ein Gridlayout ist in Abbildung 4 dargestellt.

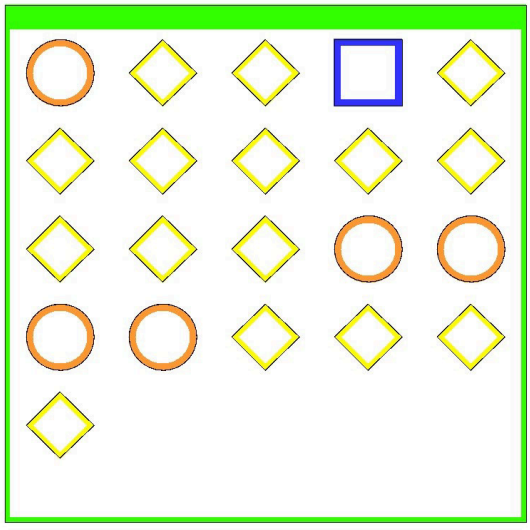


Abbildung 4 Gridlayout

Eine weitere Anordnung der Knoten ist durch das Springlayout möglich. Dabei wird ein Federkräftealgorithmus benutzt, um die Position der Knoten zu berechnen. Einzelheiten und Berechnungsformeln dieses Algorithmus sind in [97] beschrieben. In dem in Abbildung 5 gezeigten Beispiel werden

die Anzahl der Beziehungen von Konten als Kräfteeigenschaft benutzt und somit ziehen Konten mit vielen Beziehungen Knoten an, während Knoten mit wenigen Beziehungen sich eher am Rand befinden.

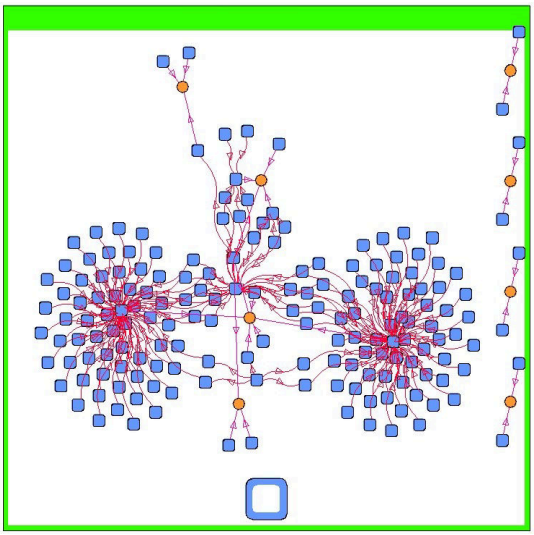


Abbildung 5 Springlayout

Das Treemaplayout wurde erstmals von B. Shneiderman in [93] erwähnt. Die Kernidee ist, dass die Größe der Knoten von der Quantität der zu visualisierenden Eigenschaft abhängt. Ein Beispiel wird in Abbildung 6 gezeigt.

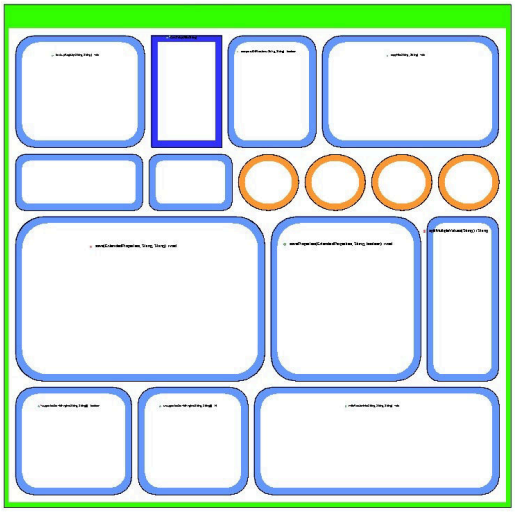


Abbildung 6 Treemaplayout

Außerdem ist ein Element von SHriMP das Treelayout. Dazu werden die Kindknoten eines Knotens auf Grund ihrer zu visualisierenden Eigenschaft in Schichten eingeteilt, so dass eine Baumstruktur entsteht. (siehe Abbildung 7)

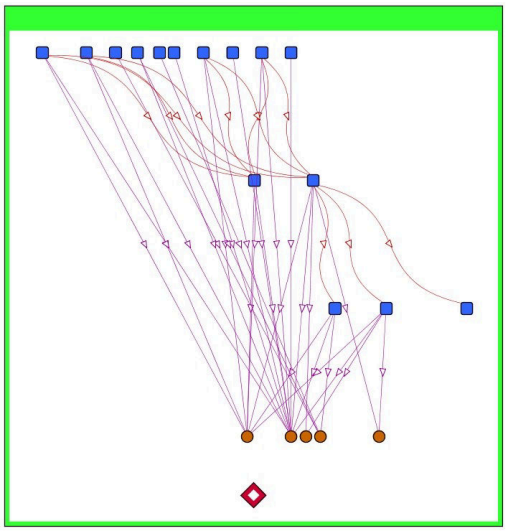


Abbildung 7 Treelayout

Die kreisförmige Anordnung wird Radiallayout genannt. Dabei werden die Elemente in konzentrischen Kreisen um die Knoten aus der höheren Ebene angeordnet. In Abbildung 8 kann man leicht erkennen, dass die Knoten der untersten Ebene den Umfang des äußeren Kreises bilden.

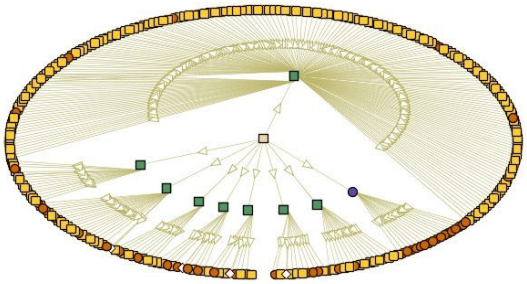


Abbildung 8 Radiallayout

2.2.4 Navigation

Die Art und Weise, wie sich Benutzer durch große Datenmengen bewegen und sie „erforschen“, hat einen sehr hohen Stellenwert in SHriMP [97]. Deshalb wurde sehr viel Wert auf die Navigation durch den verschachtelten Graphen gelegt.

Eine wesentliche Technik zur Verringerung des kognitiven Aufwandes bei der Navigation ist das Pan & Zoom - Verfahren. Dabei werden Schwenk- und Zoomaktionen derart kombiniert, dass der Benutzer stets den Kontextüberblick behält und nicht die Orientierung verliert. Wenn z.B. ein Knoten (als Quelle) über eine Kante mit einem anderen Knoten (als Ziel) aus einem anderen Elternknoten in Beziehung steht, kann man über diese Beziehung navigieren, in dem auf der Kante den Befehl „Navigiere zum Ziel“ (Go to target) aufruft. Dann wird zuerst auf die Ebene der Elternknoten herausgezoomt, um dem Benutzer den Kontext zu zeigen. Anschließend wird in Schwenkbewegung

der neue Elternknoten fokussiert und zur Ebene des Kindknoten hineingezoomt.

Das Zoomwerkzeug ermöglicht neben Vergrößern / Verkleinern auch den so genannten Fisheye-View. Bei dieser Zoomart wird der ausgewählte Knoten vergrößert, allerdings bleiben die umgebenen Knoten sichtbar. Dadurch ist es möglich, gleichzeitig Detail- und Kontextelemente zu erkennen. Ein Beispiel ist in Abbildung 9 dargestellt.

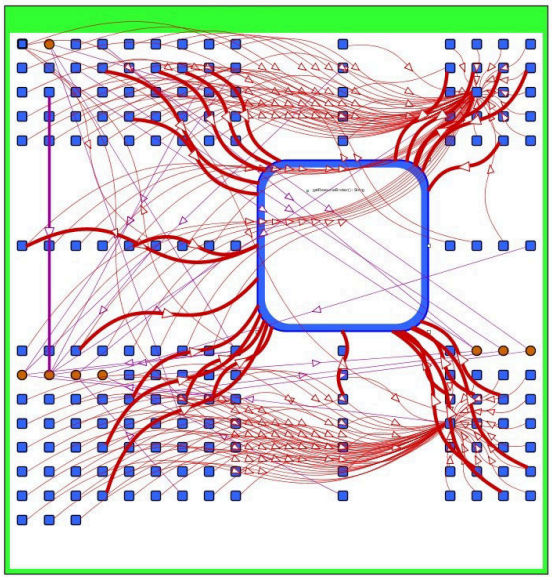


Abbildung 9 Fisheye – View

Als eine weitere Technik der Navigation wird die Hypertextmetapher benutzt. Wenn in der niedrigsten Ebene des verschachtelten Graphen Text steht, z.B. Quelltext, dann kann wie im Internet durch das Anklicken von farblich hervorgehobenen Links navigiert werden. Die Navigation selbst wird durch die oben beschriebenen Schwenk- und Zoomaktionen unterstützt.

2.3 Creole

Dieser Abschnitt stellt die Werkzeuge vor, in denen die SHriMP Technologien verwendet werden. Das am weitesten entwickelte Werkzeug, Creole, wird ausführlich beschrieben. Es wird zur Visualisierung von Softwaresystem, die in Java programmiert werden, benutzt. Die aktuellste Version von Creole ist 1.0.9, die am 6.12.2004 veröffentlicht wurde [94].

2.3.1 Geschichte der Entwicklung

Die zeitliche Entwicklung und die Abhängigkeiten zwischen den Werkversionen, die SHriMP Technologie benutzen, ist in Abbildung 10 dargestellt.

Ursprünglich war SHriMP in das Softwarevisualisierungswerkzeug Rigi [86] als spezielle Ansicht eingebettet. Da aber Rigi in C++, SHriMP dagegen in Java programmiert wurde, war die Integration nie vollständig. Deshalb entschied sich die CHiSEL Group dafür, SHriMP in einer Stand-Alone Version zu entwickeln. Das größte Problem war die Datengewinnung aus dem Informationsraum, die vorher von Rigi übernommen wurde [74].

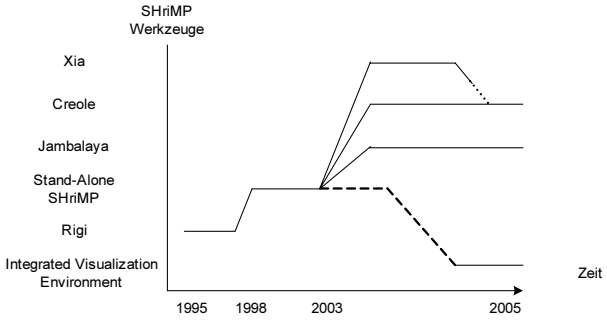


Abbildung 10 Entwicklung von SHriMP Werkzeugen

Nach dem das Datengewinnungsproblem gelöst wurde, versuchte die CHiSEL Group die Wirksamkeit der SHriMP Technologien in Teststudien nachzuweisen. Dabei wurde allerdings deutlich, dass ein Vergleich mit bestehenden Entwicklungsumgebungen der falsche Ansatz war. Vielmehr war die Frage, in wie weit die verschiedenen SHriMP Technologien den Softwareentwickler innerhalb einer Entwicklungsumgebung unterstützen können, von Interesse [64]. Deshalb wurde die Stand-Alone Architektur von SHriMP zu einer Plug-in Architektur weiterentwickelt. Wie in Abbildung 10 bei dem Jahr 2003 erkennbar ist, wurden drei Plug-ins entwickelt:

- Jambalaya
- Creole
- Xia

Jambalaya ist ein Plug-in für das Protégé Werkzeug der Universität von Stanford und wird für die Visualisierung von Ontologien verwendet [82].

Creole ist eine Plug-in für die Eclipse Plattform [27], die von der Eclipse Foundation entwickelt wird. Creole wird für die Visualisierung von Java Quelltext verwendet.

Xia ist auch ein Plug-in für die Eclipse Plattform und wurde für die Visualisierung von Eigenschaften bezüglich der Teamzusammenarbeit verwendet. Im Wesentlichen wurden Informationen aus dem Versionsverwaltungssystem CVS extrahiert, dass in der Eclipse Plattform enthalten ist. Wie in Abbildung 10 oben rechts gezeigt ist, wurde Xia kürzlich in Creole integriert [107]. Allerdings konnte bei der Evaluierung von Creole diese Integration nicht vollständig bestätigt werden.

2.3.2 Einbindung von Creole in Eclipse

Die Eclipse Plattform ist eine Entwicklungsumgebung "für alles und nichts im speziellen" [27]. Eine grundlegende Funktionalität, wie ein Editor und ein Fenstersystem wird in der Standarddistribution ausgeliefert. Durch Plug-ins, wie die *Java Development Tools (JDT)* wird die Plattform z.B. zu einer Softwareentwicklungsumgebung für Java erweitert.

Creole benutzt die Plug-in Schnittstelle von Eclipse, um eigene Fenster, im speziellen den Creole Main View und verschiedene weitere Sichten anzuzeigen. Die Information über den Java Quelltext wird dabei über die JDT API eingelesen. In Abbildung 11 wird dabei der Zusammenhang gezeigt.

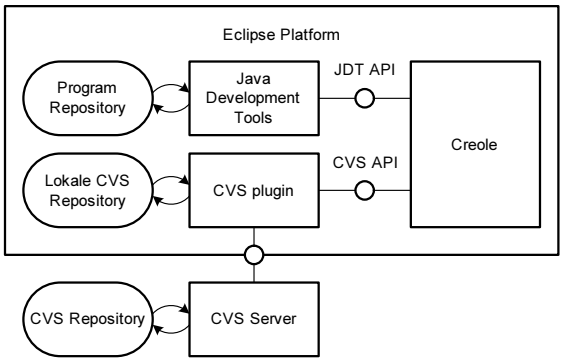


Abbildung 11 Zusammenhang zw. Eclipse & Creole

Die Informationen über die teamrelevanten Eigenschaften werden über die API des CVS-Plug-ins gewonnen.

2.3.3 Softwarevisualisierungsfunktion

Nach dem in Abschnitt 2.2 die Technologien und Konzepte von SHriMP vorgestellt wurden und in Absatz 2.3.2 die Informationsgewinnung von Creole aus dem Java Quelltext erläutert wurde, ist nun die folgende Frage zu beantworten: Wie werden diese Elemente aufeinander abgebildet? Die Funktion wird in Abbildung 12 dargestellt.

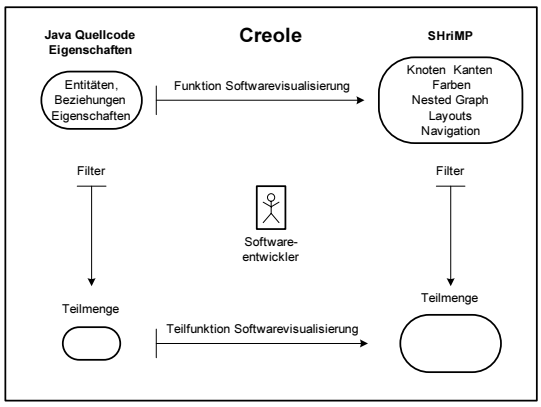


Abbildung 12 Softwarevisualisierungsfunktion

Die links stehenden Entitäten, Beziehungen, Eigenschaften sind Elemente des Informationsraumes, in diesem Fall der Java Software. Auf der rechten Seite sind die Elemente des Visualisierungsraumes Knoten, Kanten, Farben, Layouts usw. dargestellt. Die Softwarevisualisierungsfunktion bildet nun die Softwareelemente auf die Visualisierungselemente ab und definiert dadurch Creole als Softwarevisualisierungswerkzeug. Weiterhin verfügt Creole über Filter, die sowohl die Information aus dem Java Quelltext, als auch die angezeigten Knoten und Kanten einschränken können. Bei großen Datenmengen ist es die Benutzung von Filtern absolut notwendig, um ein Konzentration auf bestimmte Sachverhalte zu ermöglichen.

In Abbildung 13 ist die Zuordnung der Standardauslieferung von Creole aufgeführt. Durch den Benutzer kann diese Zuordnung verändert werden. Eine zweckmäßige Zuordnung ist in Absatz 2.4.3, verbunden mit einer Bewertung der Standardzuordnung, beschrieben.

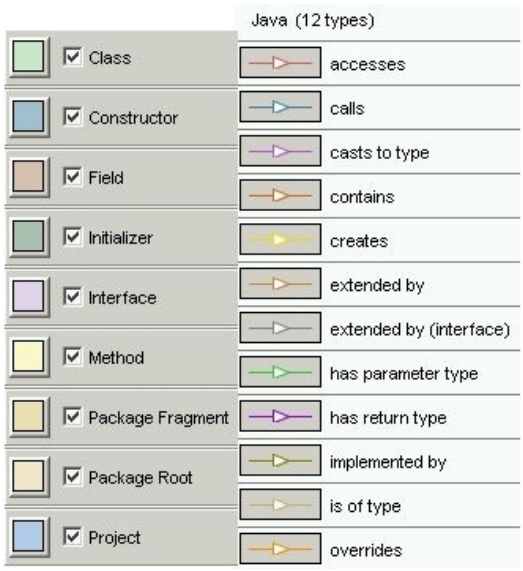


Abbildung 13 Standardzuordnung Creole

2.3.4 Vergleich zu UML

Die von Creole visualisierbaren Java Strukturen entsprechen zum größten Teil dem, in der Unified Modeling Language (Version 2.0) [105], modellierbarem Klassendiagramm. Ein Vergleich zu den im UML-Metamodell festgelegten Notationselementen mit den Softwareelementen von Creole ist in Abbildung 14 dargestellt.

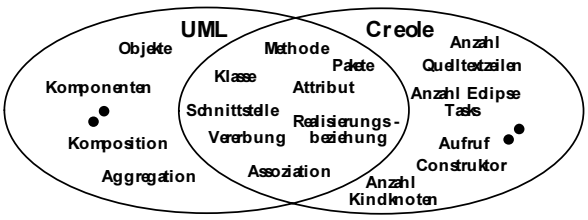


Abbildung 14 Vergleich UML mit Creole

Aus dem Klassendiagramm können alle Entitäten wie Klassen, Methoden, Schnittstellen usw. auch in Creole visualisiert werden. Einschränkungen dagegen existieren bei den Beziehungen. Die in UML bekannte Assoziation entspricht den „greift zu“ - und „ruft auf“ Beziehungen von Creole. Für die Aggregation und Komposition dagegen sind mehr Informationen notwendig, die nicht direkt aus dem Quelltext gewonnen werden können. Die Vererbungsbeziehungen und Realisierungsbeziehungen haben direkte Entsprechungen in Creole. Natürlich ist UML wesentlich mächtiger, Creole dagegen kann zusätzlich Eigenschaften wie Anzahl der Quelltextzeilen darstellen oder genauer auflösen beispielsweise das Aufrufen von Konstruktoren.

2.3.5 Creole Main View

Im Hauptfenster von Creole wird die Softwarevisualisierung dargestellt. Unterstützt wird diese Ansicht durch das Hierarchical View Fenster, in dem es zur verbesserten Kontexterhaltung beiträgt. Am Anfang wird in den Java Development Tools ein Working Set ausgewählt. Dabei handelt es sich um eine Untermenge des gerade aktuell bearbeiteten Java-Systems. Creole extrahiert dann die Informationen aus diesem Working Set und stellt sie mit den Standardeinstellungen dar. Nun kann der Softwareentwickler mit der „Erforschung“ der Systems über die Softwarevisualisierung beginnen. Als Beispiel ist in Abbildung 15 links den Hierarchical View dargestellt und rechts der Main View mit dem verschalteten Graphen bis zu einer Tiefe von drei Ebenen.

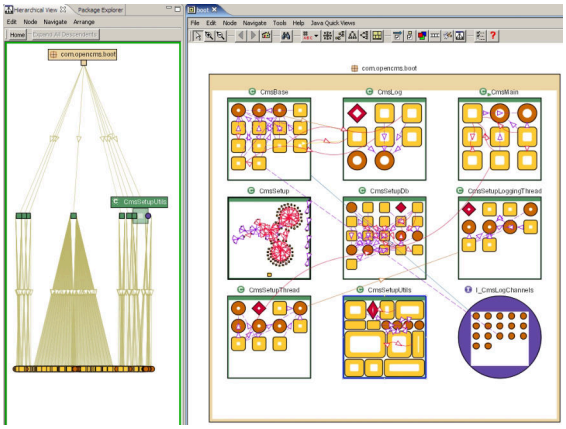


Abbildung 15 Creole Main and Hierarchical View

2.3.6 Filter

Die in Abbildung 13 gezeigten Visualisierungsmöglichkeiten können jeweils separat ein- oder ausgeblendet werden. Es gibt zwei Arten von Filtern, den Knotenfilter und den Kantenfilter. Mittels dieser Filter ist es für den Benutzer möglich, für die jeweilige aktuelle Aufgabe die optimale Visualisierung zu konfigurieren.

2.3.7 Weitere Features

Eine weitere Visualisierungsmöglichkeit von Creole ist die Zuweisung von Quelltexteigenschaften zu bestimmten Farben. In der aktuellen Version können folgende Quelltexteigenschaften visualisiert werden:

- Knotentyp (Standard)
- Anzahl der Quelltextzeilen
- Anzahl der enthaltenen Kindknoten der nächsten Ebene
- Anzahl der enthaltenen Kindknoten aller tieferen Ebenen
- Anzahl der „Eclipse – Tasks“ d.h. TODO – Kommentare im Quelltext

Über das „Attribute Panel“ kann die Zuordnung von Farben auf diese Eigenschaften und die aktuelle Auswahl erfolgen. Allerdings kann immer nur eine Eigenschaft angezeigt werden.

2.4 Analyse von OpenCMS

Zur Evaluierung von Creole wurde das frei verfügbare Content Management System OpenCMS [77] verwendet. OpenCMS hat in der Version 5.0.1 einen Umfang von 320.000 Quelltextzeilen. Die Entwicklung wurde 2000 begonnen und basierend auf den CVS-Informationen arbeiteten insgesamt 31 Programmierer an der Entwicklung. Somit handelt es sich bereits um ein Softwaresystem von respektabler Größe, so dass das Erlangen des Systemverständnisses nicht mehr praktikabel über das Lesen des Quelltextes erzielt werden kann. Dem Autor war OpenCMS völlig unbekannt, so dass Kontextwissen ausgeschlossen werden konnte.

2.4.1 Vorgehen bei der Analyse

Nach dem Herunterladen des Quelltextes wurde durch Eclipse ein neues Java-Projekt angelegt. Der Projektwizard der Java Development Tools ist sehr leistungsfähig, so dass bis auf die Angabe des Dateipfades keine zusätzlichen Anpassungen vorgenommen werden mussten. Die Installation von Creole war ebenso einfach. Nach dem Herunterladen der Distribution entpackt man die Binarys in den Eclipse Ordner und startet die Eclipse Plattform. Anschließend ist die Creole Perspektive verfügbar oder für jede Java Klasse kann per Rechtsklick im Kontextmenü „Navigate to Creole Main View“ aufgerufen werden. Nach dem Festlegen des Working Sets kann mit der Erforschung von OpenCMS begonnen werden.

2.4.2 Mögliche kognitive Strategien

Der Autor hat bei der Analyse von OpenCMS intuitiv zwei Strategien benutzt, die in [97] als Bottom-up und As-needed Strategien bezeichnet werden. Die erste Strategie beschreibt das Vorgehen beim Erlangen des Systemverständnisses durch Lesen des Quelltextes und anschließendem Zusammensetzen von Information zur Gewinnung von Informationen auf höherer Abstraktionsebene. Die zweite Strategie beschreibt das Verfolgen von Kontroll- und Informationsfluss im, für die aktuelle Aufgabe relevanten, Systembereich. Allerdings wird bei dieser Strategie nur ein teilweises Systemverständnis erlangt, da nicht relevante Systemteile nicht betrachtet werden.

2.4.3 Ergebnisse

Beide in Abschnitt 2.4.2 verwendeten Strategien wurden von Creole optimal unterstützt. Bei der Bottom-Up Strategie war die Navigation durch die Schwenk- und Zoomaktionen sehr einfach und der Kontext war ohne Mehraufwand immer präsent. Zur Unterstützung der As-needed Strategie war die Benutzung des Query Views von großem Vorteil. Der Query View ermöglicht die Namenssuche nach Java Knotentypen und zeigt das Ergebnis als graphische Präsentation mit den in der Umgebung liegenden Knoten an. Als Beispiel ist das Ergebnis zur „main“ - Methode in Abbildung 16 dargestellt. Mit Hilfe dieser Darstellung ist das Verfolgen von Kontroll- und Datenstrukturen sehr einfach.

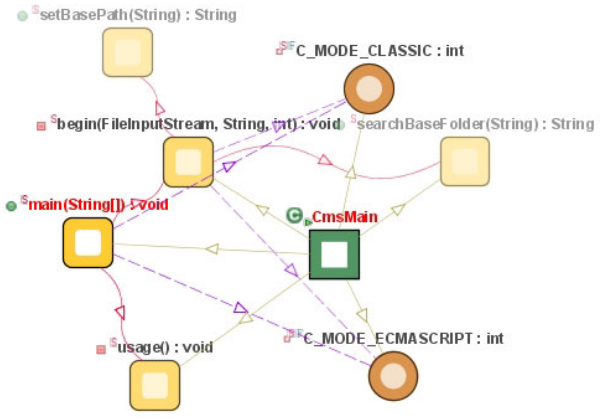


Abbildung 16 Query View "main"

Die in Abbildung 13 gezeigte Standardzuordnung der Java Elemente zu den Visualisierungselementen erwies sich als nicht zweckmäßig. Da alle Knoten durch die gleiche geometrische Figur (Quadrat) und die Kanten alle durch den geraden durchgehenden Pfeil visualisiert wurden, war die Unterscheidbarkeit der Elemente sehr gering. Dieses Problem wurde durch die kontrastarmen Farben noch verstärkt. Der Autor wählte deshalb für alle Knoten und Kanten soweit möglich unterschiedliche Formen und kontraststarke Farben. Als sehr zweckmäßig erwies sich die Verwendung der Java Development Tool Farbcodierung, z.B. grün als Klassenfarben und violett für Interfaces. Für Eclipsebenutzer ist diese Visualisierung sehr intuitiv.

Das Hauptproblem bei der Analyse von OpenCMS mit Creole lag in der Performance. Bezüglich der Maßeinheiten von Creole hatte OpenCMS einen Umfang von 9000 Java Entitäten und 200.000 Beziehungen zwischen diesen Entitäten. Auf der zur Verfügung stehenden Testhardware (Intel Pentium 4, 1.4 GHz, 768 MByte RAM) musste Eclipse der gesamte Arbeitsspeicher per Kommandozeilenbefehl (-vmargs –Xmx768) zugewiesen werden, um ein Laden des gesamten Quelltextes zur ermöglichen. Die Reaktionszeiten von Creole waren sehr langsam, so dass ein flüssiges Arbeiten nicht möglich war.

Deshalb wurde die Analyse auf das `com.opencms.boot` – Package beschränkt. Es besteht aus 8 Java-Klassen und einem Java-Interface. Creole hat insgesamt 300 Knoten visualisiert, die in diesem Fall 2655 Quelltextzeilen entsprechen. Bei dieser Größe war sowohl interaktives Navigieren und Analysieren, als auch der schnelle Aufbau und Berechnung verschiedenster Layouts möglich. Das Boot-Package ist auch Abbildung 15 in dargestellt. Mit Hilfe der Visualisierung konnte sehr schnell ein Verständnis des Boot-Packages erlangt werden und die wichtigen Klassen bzw. Methoden identifiziert werden.

2.5 Einordnung und Bewertung

2.5.1 Abbildungsmächtigkeit in Bezug auf Systemmodelle vom UML

Creole kann die Elemente der statischen Systemarchitektur, vor allen Dingen im Umfang des Klassendiagramms abbilden. Zu den Entitäten gehören Pakete, Klassen, Interfaces, Attribute, Methoden und Exceptions. Bezüglich der Beziehungen können Vererbung und Assoziation direkt dargestellt werden. (Siehe auch Abbildung 14)

Elemente der dynamischen und physischen Systemarchitektur, wie Aufrufketten oder Bibliotheken-Strukturen können nicht abgebildet werden.

2.5.2 Klassifikation des Software-Visualisierungsansatzes

Das zentrale Visualisierungsprinzip ist die Darstellung von hierarchisch strukturierten Daten und ihrer Zusammenhänge in einem geschachtelten Graphen mit zusammengesetzten Knoten und Kanten.

Der Visualisierungsprozess beginnt damit, dass die Informationen über den Java Quelltext von der JDT-API zur Verfügung gestellt werden. Creole liest diese Information dann ein. Anschließend wird entsprechend der gesetzten Einstellungen die Softwarevisualisierung aufgebaut und angezeigt. Siehe auch Abbildung 11.

Die ursprüngliche Intention war, Creole bei der Erstanalyse eines Systems einzusetzen. Durch die Integration in Eclipse haben sich die Einsatzmöglichkeiten stark erweitert. Creole kann nun zusätzlich bei der Weiterentwicklung von Software benutzt werden. Wenn die Integration von Xia abgeschlossen ist, kann es auch zur Analyse von inkrementellen Änderungen benutzt werden [65,107].

Zur Zeit wird nur die Visualisierung von Java Quelltext der Java Versionen 1.4.2 und 1.5 unterstützt [94].

Creole ist völlig unabhängig von bestimmten Programmierkonventionen. Es müssen keinerlei zusätzliche Elemente in den Quelltext eingefügt werden.

Die Skalierbarkeit in Bezug auf die Systemgröße ist problematisch. Durch die Verwendung des Eclipse Working Set-Prinzips und den Einsatz von Visualisierungsfiltern ist zwar ein sehr fein einstellbares Herunterskalieren möglich, aber ab einer Systemgröße von ca. 3000 Quelltextzeilen ist kein flüssiges Arbeiten mehr möglich.

Creole verfügt über einen sehr hohen Automatisierungsgrad. Zur Anzeige der Visualisierung ist lediglich die Auswahl eines geeigneten Working Sets notwendig.

Mit SHriMP wurden bei Evaluierungsstudien Systeme wie ein Monopolyspiel [98] und SHriMP selbst analysiert [65]. Zu Creole selbst existieren noch keine Referenzen auf untersuchte Softwaresysteme.

2.5.3 Werkzeugeigenschaften

Creole ist als Plug-in in die Eclipse Plattform ab Version 3.0 eingebettet. Es handelt sich dabei um ein freiverfügbares Eclipse Plug-in, genauere Lizenzbedingungen sind nicht bekannt. Der Quelltext ist allerdings nicht frei verfügbar.

Folgende Plattformen werden unterstützt [94]:

- Windows mit Java 1.4.2 oder Java 1.5
- Linux mit Java 1.5 (nur bestimmte Linux Versionen)
- Mac wahrscheinlich ab Eclipse 3.1.0 (noch nicht veröffentlicht) mit Java 1.5

2.5.4 Forschung und Entwicklung

Folgende Institutionen sind an der Entwicklung von Creole beteiligt:

- CHiSEL Group, Universität von Victoria, Victoria, Kanada
- Kooperationen mit der Universität von Stanford, USA für das Protégé Plug-in

An der Forschung sind Margaret-Anne Storey, wissenschaftliche Mitarbeiter und Masterstudenten der CHiSEL Group beteiligt.

2.6 Schlussfolgerungen

Die SHriMP Technologie und Konzepte konzentrieren sich auf die Navigation und Erforschung von visualisierten Informationsräumen unter Reduzierung von kognitivem Mehraufwand.

Bei Creole ist dabei die Struktur des Java Quelltextes von Interesse. Durch die Einbettung in Eclipse wurde das Problem der Informationsgewinnung gelöst und die Verwendbarkeit durch den Benutzer auf Grund der direkten Benutzbarkeit innerhalb einer Entwicklungsumgebung entscheidend erhöht.

Gerade die in [97] beschriebenen kognitiven Strategien werden sehr gut unterstützt. Für den Einsatz der Bottom-up und Top-down Strategie ist das Navigationskonzept und die Darstellung im verschachtelten Graphen sehr zweckmäßig. Die As-needed und Knowledge-based Strategien werden durch die sehr effizienten Suchmöglichkeiten im „Query View“ und durch die Kontexterhaltung im Hierarchical bzw. Thumbnail View optimal unterstützt.

Die Visualisierungskonzepte sind hervorragend umgesetzt und ermöglichen ein effektives Arbeiten mit Creole. Die Abbildung der Java Elemente auf die Visualisierungselemente wird allerdings vernachlässigt und es ist dem Benutzer selbst überlassen, eine für ihn geeignete Visualisierungsfunktion zu finden.

Die größten Probleme bestehen zur Zeit noch in der Verarbeitung von großen System mit viel Quelltext. Allerdings kann dieses Problem durch größeren Hardwareeinsatz gemindert und in Zukunft, entsprechend des Moore’schen Gesetzes, gelöst werden.

Zur Weiterentwicklung von Creole ist seitens der CHiSEL Group nicht viel bekannt. Die Integration von Xia wird wahrscheinlich im nächsten Release vollendet werden. Auch die Visualisierung von dynamischen Eigenschaften über die Java Virtuell Machine Profiler API und anderer Programmiersprachen ist denkbar.

KONZEPTE DER SOFTWAREVISUALISIERUNG
FÜR KOMPLEXE, OBJEKTORIENTIERTE
SOFTWARESYSTEME

Kapitel 3

SHriMP/Creole in der Anwendung

Nebojsa Lazic

Kapitel 3: SHriMP/Creole in der Anwendung

Nebojsa Lazic

Zusammenfassung. *Das folgende Paper beschreibt den Einsatz des Softwarevisualisierung-Tools SHriMP/Creole bei der Analyse bzw. beim Verstehensprozess des Softwaresystems jEdit*

3.1 Einführung

Softwarevisualisierung-Tools unterstützen beispielsweise den Entwickler eines Softwaresystems oder den Manager eines Softwareprojekts beim Verstehensprozess der betrachteten Softwarefragmente.

SHriMP/Creole [99,8,100,64,94] ist, nach den Ausführungen seiner Entwickler [13], insbesondere für das „Reverse Engineering“ eine ausgesprochen gute Hilfe. Eine der Hauptfunktionalitäten von SHriMP/Creole ist die Exploration der Struktur des betrachteten Softwaresystems, indem es beispielsweise über die Navigations-funktionalitäten „Magnify-In“/„Magnify-Out“ das jeweils selektierte Element fokusiert, und die enthaltenen Subelemente („Nested View“) vergrössert bzw. verkleinert anzeigt.

SHriMP/Creole macht im Rahmen der Visualisierung Gebrauch von der Zooming-Bibliothek Piccolo [79].

Das im Rahmen des Seminars „Software Visualization“ am HPI zu analysierende Softwaresystem ist jEdit [49].

3.2 jEdit

jEdit ist ein in Java implementierter „Open Source” (GNU General Public License [38]) Cross-Plattform Programmier-Text Editor.

Das Softwaresystem ist über einen Entwicklungszeitraum von 7 Jahren entstanden. Die aktuelle Version (25.01.2005) ist 4.3pre1, für die Analyse mittels SHriMP/Creole wurde die Version 4.2stable zugrunde gelegt.

Die Besonderheiten von jEdit werden durch seine Community wie folgt zusammengefasst [50]:

> *„Combines the best functionality of Unix, Windows and MacOS text editors“*

Für einen Überblick seien davon genannt:

- unbegrenztes „Undo/Redo“
- unbegrenzte Anzahl an „Clipboards“
- „Kill Ring“ speichert/erkennt automatisch zuvor gelöschten Text
- „Marker“ markieren Positionen in Dateien für ein späteres Wiederauffinden
- „Syntax Highlighting“ für über 130 Dateitypen (C, C++, Java, C#, Latex, etc.)
- mehr als 80 Plugins [51]

Eine vollständigere Auflistung der Funktionalitäten ist unter [50] gegeben. Ausführlichere Informationen zu jEdit sind unter [49] zu finden.

Einige wichtige Metriken bezüglich jEdit sind in Tabelle 1 zusammengefasst.

3.2.1 Exploration der Paketstruktur jEdit

Eine der Stärken von SHriMP/Creole ist die Exploration der Struktur eines vorliegenden Softwaresystems.

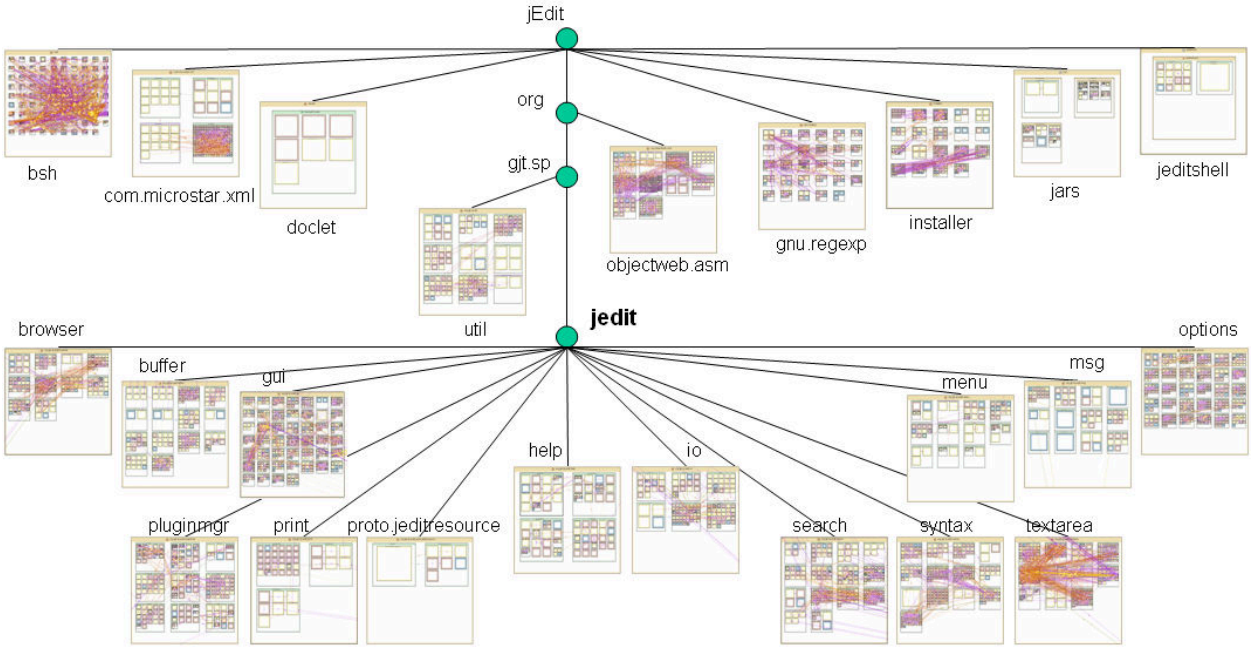


Bild 1
Paketstruktur (Hauptpakete) von jEdit

Ein Softwaresystem ist, insbesondere objektorientierte Softwaresysteme, hierarchisch strukturiert. Eine resultierende Darstellung wäre daher ein hierarchischer Strukturbaum, der in Bild 1 zu sehen ist. Der Strukturbaum ist in diesem Fall Schritt für Schritt durch Exploration jedes einzelnen Pakets mittels SHriMP/Creole entstanden, was einen enormen Zeitaufwand bedeutete; die grafische Darstellung von Bild 1 ist mittels „MS PowerPoint“ erstellt worden.

SHriMP/Creole bietet zwar eine automatische Generierung des „Hierachical View“, diese Darstellung ist allerdings derzeit nicht brauchbar, sodass die Entscheidung gefällt wurde, für den Verstehensprozess einen eigenen Strukturbaum (Bild 1) zu erstellen.

Für die Analyse von jEdit mittels SHriMP/Creole standen 2 PC-Konfigurationen zur Verfügung:

1. AMD 800 MHz / 256 MB RAM, Eclipse 3.0.1 mit Creole 1.0.9 unter Windows XP Professional
2. AMD 1800+ MHz / 512 MB RAM, Eclipse 3.0.1 mit Creole 1.0.9 unter Windows XP Professional

Während der anfänglichen Experimentierphase mit SHriMP/Creole stellte sich heraus, dass PC-Konfiguration 1 aufgrund des langsamen Prozessors und geringem Arbeitsspeicher für die Analyse nicht in Frage kommt.

Mit PC-Konfiguration 2 liess sich besser arbeiten, aber in der Summe ist auch diese Konfiguration nur bedingt tauglich, da von der Vielzahl der angebotenen Funktionalitäten nur mit „Magnify“ vernünpftig gearbeitet werden konnte, ebenfalls konnte nur der „Nested View“ genutzt werden.

In Bild 1 ist die Struktur von jEdit deutlich zu erkennen.

Neben den jEdit Hauptpaketen

> browser, buffer, gui, help, io, menu, msg,
> options, pluginmgr, print,
> proto.jeditresource, search, syntax, textarea

die im Paket jedit zusammengefasst sind, werden mit der Version 4.2 auch die Pakete

> bsh, com.microstar.xml, doclet,
> gnu.regexp, installer, jars, util,
> objectweb.asm

ausgeliefert, die erweiterte Funktionalitäten wie BeanShell (bsh), die Verarbeitung von XML (com.microstar.xml) oder das Nutzen von „Regular Expressions“ (gnu.regexp) bereitstellen.

Die API Dokumentation [53] beschreibt die Funktionalitäten der einzelnen Pakete genauer.

Die in Bild 2 gegebene Sicht auf den root-Knoten der jEdit-Pakethierarchie gibt Anlass zur Diskussion:

1. in der anfänglichen Experimentierphase mit Eclipse, SHriMP/Creole und dem Quellcode von jEdit, ergab sich der Sachverhalt, dass durch den eher zufällig eingeschalteten Eclipse „Package Explorer“ Pakete lokalisiert wurden (doclet, jars, jeditshell), in denen Quellcodedateien vorhanden sind, die sowohl syntaktische Quellcodefehler enthalten, als auch Pakete importieren, die in der ausgelieferten Version 4.2 nicht enthalten sind.

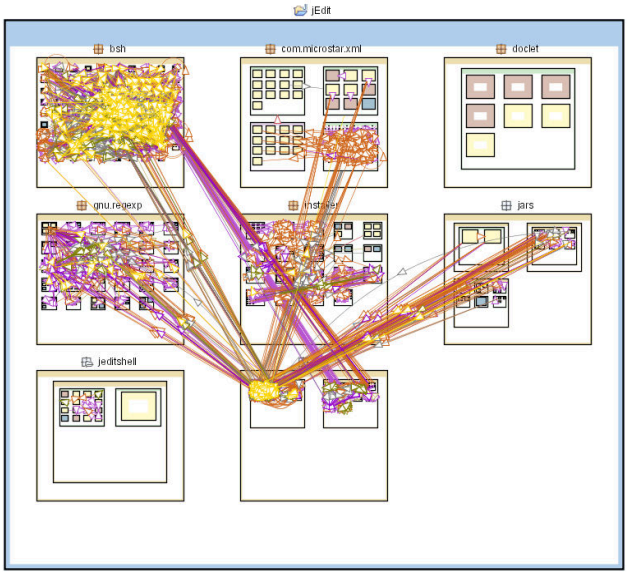


Bild 2
Sicht auf den root-Knoten der jEdit-Pakethierarchie

SHriMP/Creole zeigt diese Fehler weder auf Quellcodeebene noch auf visueller Ebene explizit an. In Bild 2 könnte man allerdings durch die Auffälligkeit, dass Verbindungen zu einzelnen Paketen fehlen, beispielsweise zu doclet, auf derartige Fehler im Quellcode schliessen.

2. der Eclipse „Package Explorer“ gibt beim Expandieren des root-Knotens insgesamt 14 Pakete preis, wobei SHriMP/Creole selbst nur diejenigen Pakete visualisiert (8 Packete, Bild 2), die tatsächlich Java Quellcodedateien enthalten. Wünschens-wert wäre aber, dass für das Verständnis des Gesamtsystems jEdit, alle vor-kommenden Elemente dargestellt werden.

Für das Verständnis eines Softwaresystems ist es ebenfalls von Wichtigkeit über gewisse Metriken ein Gefühl für die Komplexität des vorliegenden Softwaresystem zu gewinnen. Interessante Metriken wären beispielsweise

- Anzahl der Quellcodezeilen (ohne Kommentare und Leerzeilen), „Lines of Code (LOC)“
- Anzahl der Zeilen (inklusive Kommentare und Leerzeilen), „Number of Lines (NOL)“
- Anzahl der Pakete (bilden die Struktur), „Number of Packages (NOP)“

Die Liste der interessanten Metriken kann beliebig erweitert werden, und ist abhängig von der Leistungsfähigkeit des Tools, welches die Metriken liefert.

SHriMP/Creole hat einen integrierten „Attribute Viewer“, der für ein selektiertes Element beispielsweise die LOC oder auch die Anzahl der Subelemente liefern soll. Diese Funktionalität ist derzeit nicht fehlerfrei

jEdit – LOC: 88.443 NOL: 140.666 NOP: 72				
bsh LOC: 17.404 NOL: 26.827	**com.microstar.xml** LOC: 2.503 NOL: 4.899	**doclet** LOC: 66 NOL: 101	**gnu.regexp** LOC: 2.044 NOL: 4.234	**installer** LOC: 5.153 NOL: 7.486
jars LOC: 1.321 NOL: 2.113	**jeditshell** LOC: 149 NOL: 350	**util** LOC: 1.014 NOL: 1.768	**objectweb.asm** LOC: 2.249 NOL: 4.971	
jedit LOC: 15.455 NOL: 25.994	**browser** LOC: 3.518 NOL: 4.848	**buffer** LOC: 2.172 NOL: 3.594	**gui** LOC: 10.146 NOL: 14.859	**help** LOC: 1.116 NOL: 1.560
io LOC: 1.420 NOL: 2.571	**menu** LOC: 1.093 NOL: 1.668	**msg** LOC: 262 NOL: 918	**options** LOC: 4.321 NOL: 5.991	**pluginmgr** LOC: 2.513 NOL: 3.402
print LOC: 495 NOL: 713	**proto.jeditresource** LOC: 107 NOL: 165	**search** LOC: 3.063 NOL: 4.808	**syntax** LOC: 2.246 NOL: 3.563	**textarea** LOC: 8.613 NOL: 13.263

Tabelle 1
jEdit Metriken, ermittelt mittels CodePro AnalytiX [16]

implementiert und kann somit nicht genutzt werden.

Stattdessen wurde zur Metrikgewinnung auf eins der auf dem Markt verfügbaren und ohne grossen Aufwand auch ausführbaren Tools zurückgegriffen: „CodePro AnalytiX“ [16]. Die Ergebnisse sind in Tabelle 1 zusammengefasst.

3.2.2 jedit.jEdit.main()

Nachdem die Struktur eines Softwaresystems erarbeitet und sowohl visuell als auch mental erfasst wurde, kommt es im zweiten Schritt darauf an, zu verstehen, wie die Software „funktioniert“. Man sucht nach dem Einstiegspunkt, d.h. nach der Hauptklasse des Systems, in der die main()-Methode definiert ist.

Weiterhin sind manuell, ebenfalls in zeitintensiver Arbeit, diejenigen Verbindungen innerhalb der Klasse jEdit gefiltert worden, die direkt von der main()-Methode ausgehen (Bild 3, Element rechts). Da die main()-Methode diejenige ist, die als erste ausgeführt wird, ist es von besonderem Interesse für das Verständnis des Softwaresystems herauszufinden, was in der main()-Methode passiert, und das ist mit der durch SHriMP/Creole gegebenen visuellen Darstellung besser möglich, als durch die Sicht auf den reinen Quellcodetext.

Gelangt man wie in Bild 3 auf die Sicht eines Blattknotens, in dem Fall der Klasse jEdit, so besteht dann die Möglichkeit sich auch den Quellcode anzeigen zu lassen.

Wiederum stellt man dann fest, dass

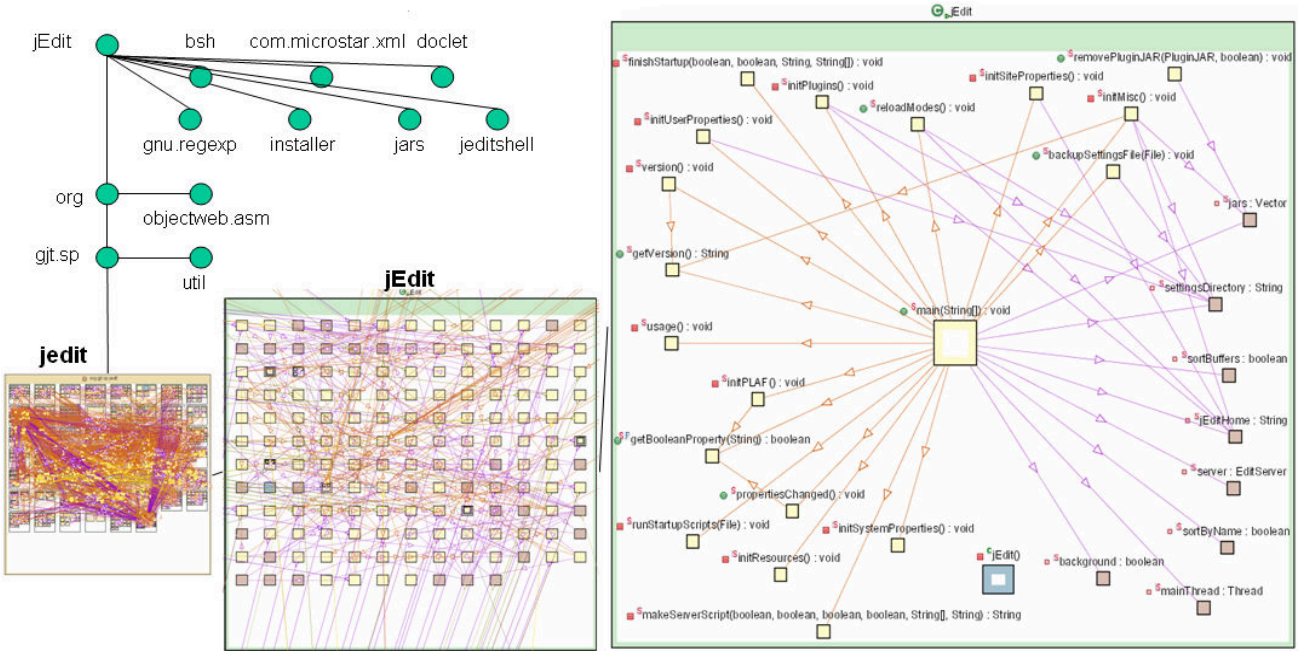


Bild 3
main()-Methode und ihre Aktivität in der Hauptklasse jEdit

SHriMP/Creole bietet eine Suchfunktion, mit der man gezielt nach Elementen suchen kann. Mit Hilfe dieser Suchfunktion ist die Hauptklasse jEdit und ihre main()-Methode identifiziert worden (Bild 3); allerdings enthält jEdit einige Klassen die main()-Methoden enthalten, was nicht unüblich ist für komplexe Softwaresysteme, sodass aufgrund der unzureichenden Annotation der Ergebnismenge, ein Ergebniseintrag nach dem anderen ausprobiert werden muss, bis letztendlich die richtige main()-Methode gefunden wird.

Klar wird hierbei der Nachteil: bei einigen wenigen Ergebniseinträgen wäre der User noch bereit nach dem richtigen Ergebniseintrag zu suchen, bei einigen Hundert sicherlich nicht mehr.

SHriMP/Creole auch bei der Visualisierung der Verbindungen der Elementen untereinander nicht alle Verbindungen darstellt. Es fehlen etliche Verbindungen, beispielsweise diejenigen, die die main()-Methode durch Aufrufe statischer Methoden diverser Klassen des Softwaresystems jEdit eingeht.

Erneut wird durch diese unvollständige Visualisierung der Mehrwert des Einsatzes eines Softwarevisualisierungssystemen, in diesem Fall SHriMP/Creole, für den Verstehensprozess in Frage gestellt.

3.3 Zusammenfassung

Die von den Entwicklern von SHriMP/Creole angegebenen Möglichkeiten (verschiedene Views und Navigationsfunktionalitäten) eignen sich grundsätzlich für den Einsatz im Verstehensprozess eines Softwaresystems.

Problematisch ist allerdings die tatsächliche effektive Arbeitsweise mit diesem Tool. Die Chisel Group empfiehlt auf ihrer Webseite einen PC mit 700 MHz/512 MB RAM, was allerdings nur für kleine Softwaresysteme mit einigen wenigen tausend Quellcodezeilen ausreichen dürfte.

Viele der angebotenen Funktionalitäten, in der Regel die verschiedenen Views (z.B. „Hierarchical View“) konnten nicht genutzt werden, sei es aufgrund geringer Leistungsfähigkeit des eingesetzten PCs oder durch eine in Frage gestellte Realisierung des jeweiligen View und damit eines geringen Mehrwerts für den Verstehensprozess.

Lediglich der „Nested View“ lies sich, allerdings unter grossem Zeitaufwand, zur Analyse des Softwaresystems nutzen.

Erst durch die Zuhilfenahme weiterer Hilfsmittel der Eclipse IDE sind Mängel in der Visualisierung seitens SHriMP/Creole aufgefallen:

- nicht alle Elemente des Softwaresystems werden visualisiert, gänzlich fehlen Elemente ohne Java-Quellcode
- das Fehlen von Paketen und syntaktische Quellcodefehler werden von SHriMP/Creole nicht angezeigt

Erst die Sicht auf den Quellcode einer gewählten Klasse, in diesem Fall der Hauptklasse jEdit, und die Analyse der dort enthaltenen main()-Methode, offenbaren einen weiteren Mangel:

- nicht alle Verbindungen zwischen den Elementen werden visualisiert, z.B. Aufrufe statischer Methoden diverser Klassen fehlen gänzlich

Der Mehrwert der Suchfunktionalität könnte gesteigert werden, in dem die Ergebnismenge besser aufbereitet dem User präsentiert werden würde, um zeitaufwendiges Durchforsten der Ergebnismenge nach dem richtigen Ergebniseintrag zu vermeiden. Bei Suchanfragen mit grossen Ergebnismengen wäre ein Druchforsten ohnehin dem User nicht zu zumuten..

Weiterhin können derzeit einfachste Metriken wie LOC, NOL und NOP durch SHriMP/Creole nicht ermittelt werden, der integrierte „Attribute Viewer“ ist fehlerhaft und daher nicht nutzbar.

Würden die Entwickler von SHriMP/Creole die genannten Mängel beseitigen und stünde zur Analyse des jeweiligen Softwaresystems ein PC mit hoher Leistungsfähigkeit zur Verfügung, so ist SHriMP/Creole sicherlich ein hilfreiches Werkzeug für den Einsatz im Verstehensprozess eines Softwaresystems, und verleiht der Softwarevisualisierung einen höheren Stellenwert.

Es muss allerdings auch erwähnt werden, dass die Softwarevisualisierung insgesamt gesehen Gegenstand aktueller Forschung ist, somit sollte die derzeitige Funktionalität von SHriMP/Creole, ob nur eingeschränkt oder mit Mängeln versehen, nicht überbewertet werden. Die Entwickler [13] versprechen Verbesserungen in den nachfolgenden Versionen.

Kapitel 4

Metrikbasierte Softwarevisualisierung mit dem Reverse-Engineering-Werkzeug CodeCrawler

Daniel Brinkmann

Kapitel 4: Metrikbasierte Softwarevisualisierung mit dem Reverse-Engineering-Werkzeug CodeCrawler

Daniel Brinkmann

Zusammenfassung. Softwarevisualisierung ist eine Art der Informationsvisualisierung, die statische und dynamische Eigenschaften von Software grafisch aufbereitet. Diese Arbeit stellt theoretische Grundlagen der Softwarevisualisierung, Informationstypen und Polymetrische Sichten vor. Es werden die Analyse von Software, die Informationsaufbereitung sowie die grafische Präsentation beschrieben. Ein Aspekt ist die Integration mehrerer Informationen in einzelne Darstellungen. CodeCrawler, ein Werkzeug zum Reverse-Engineering objektorientierter Software, stellt die Grundlagen dieser Arbeit. Der zweite Teil befasst sich anhand einer Analyse eines Softwaresystems mit der praktischen Untersuchung von CodeCrawler. Abschließend werden die vorgestellten Methoden bewertet und in den Softwareengineering Kontext eingeordnet.

4.1 Einführung

Die Informationsvisualisierung bereitet abstrakte Informationen durch Computerunterstützung auf. Dies können sichtbare und unsichtbare Eigenschaften von sowie Beziehungen zwischen jeglichen Objekten sein. Die computererzeugte Repräsentation der Informationen ist als "wahrnehmbar" zu beschreiben, meist jedoch grafischer Natur. Dem Benutzer wird so geholfen, die Informationen leichter zu verstehen. Beispiele für Informationsvisualisierungen sind Wahlanalysen, Verläufe von Börsenkursen oder Besucherzahlen von Internetseiten.

Die Softwarevisualisierung ist eine Spezialisierung der Informationsvisualisierung. Die betrachteten Eigenschaften sind aus dem Kontext der Softwareentwicklung und beschreiben z.B. Konstrukte und Eigenschaften des Programmquellcodes. Sie kann noch weiter für Paradigmen wie die Objektorientierung spezialisiert werden.

In dieser Arbeit wird CodeCrawler untersucht, ein Programm zur Visualisierung objektorientierter Softwaresysteme. Die zentrale Arbeit an dem Werkzeug hat Prof. Dr. Michele Lanza [60,62] an der Universität Bern geleistet. Aktuell ist er als Professor an der Universität Lugano beschäftigt.

CodeCrawler entstand aus der wachsenden Bedeutung des Reverse Engineerings sogenannter Legacy-Systeme, die trotz aktueller Paradigmen wie der Objektorientierung entstehen. Dies folgt daraus, dass trotz bekannter Vorzüge oftmals keine angemessene Analyse und Design der Systeme betrieben wird. Die Systeme sind von anfang an unflexibel und können als Legacy-Systeme bezeichnet werden.

Unter Beachtung der für einzelne Personen unüberschaubaren Größe der Systeme sind Werkzeuge unerlässlich, die die Analyse von Systemen unterstützen. Oftmals nur unzureichende Dokumentation erschweren die Weiterentwicklung und Wartung erheblich, wodurch Reverse Engineering immer unverzichtbarer in der Softwareentwicklung wird.

FAMOOS[1], ein europäisches Projekt das sich mit dem Reengineering objektorientierter Software befasste, wurde aus den gleichen Gründen 1996 gestartet. Es entstand ein Meta-Modell zum Informationsaustausch zwischen Reengineering-Werkzeugen, FAMIX[2]. Es ist als XML-DTD verfügbar. In diesem Rahmen entstand unter anderem auch CodeCrawler.

Solche Werkzeuge haben folgende benutzerunterstützende Aufgaben:

- Verständnis von Strukturen entwickeln
- Verständnis des Laufzeitverhaltens gewinnen
- Auffinden von "kritischen Punkten"
- Angemessene grafische Präsentation der Ergebnisse

4.2 Theoretische Grundlagen von CodeCrawler

Im diesem Abschnitt werden die unterschiedlichen Informationstypen sowie die bei CodeCrawler verwendete Visualisierungsmethode der Polymetrischen Sichten beschrieben. Zudem werden grundlegende Analyseverfahren vorgestellt.

[1] Framework-based Approach for Mastering Object-Oriented Software Evolution
[2] FAMOOS Information EXchange Model

Die Softwarevisualisierung befasst sich allgemein mit messbaren Softwareeigenschaften wie z.B. die Anzahl der Codezeilen (Lines Of Code, LOC) oder die Anzahl von Prozeduren. Beziehungen zwischen Prozeduren und/oder Variablen des Programms werden auch erfasst. Neben den statischen können auch dynamische Aspekte verwendet werden, die die Ausführung charakterisieren.

Die objektorientierte Softwarevisualisierung erweitert dieses Modell um objektorientierte Aspekte wie die Anzahl der Klassen, deren Anzahl von Methoden (NOM) oder von Attributen (NOA). Weitere Beziehungen stellen die Vererbungshierarchien der Klassen oder "Kollaborations-Analysen" dar, die die Zusammenarbeit von Objekten erfassen.

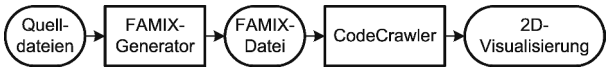


Abbildung 17: Workflow der Visualisierung.

Abbildung 17 zeigt die vereinfachten Schritte zur Visualisierung mit CodeCrawler. Die gegebenen Quelldateien werden vorverarbeitet und in einem Zwischenformat (FAMIX) abgelegt. Es enhält die Elemente des Softwaresystems und berechnete Maße, die diese beschreiben. Es wird von CodeCrawler geladen und für die Visualisierung verwendet.

4.2.1 Informationstypen in der objektorientierten Softwarevisualisierung

Dieser Abschnitt betrachtet die unterschiedlichen Informationen über ein Softwaresystems näher.

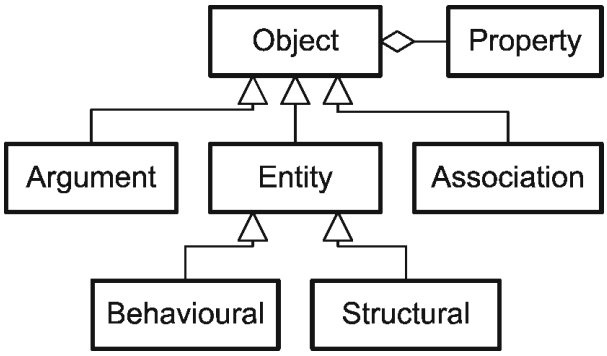


Abbildung 18: Basis des FAMIX-Modells nach [20].

Abbildung 18 zeigt die Basis des FAMIX-Modells. Elemente sind dabei immer Objekte, die Eigenschaften haben können. Die in FAMIX verfügbaren Elemente erben von den hier dargestellten Elementen, wie z.B. Klasse von "Entity", Methode von "Behavioural" oder Attribut von "Structural". Das vollständige Modell kann [20] entnommen werden. Da es nicht für alle Programmiersprachen optimiert ist, ist es erweiterbar [33]. CodeCrawler nutzt dieses Meta-Modell, weswegen die verfügbaren Informationen hiervon abhängen.

Eine einfache FAMIX-Datei enthält neben den ermittelten Klassen "<FAMIX.Class ..." auch Informationen über die Attribute "<FAMIX.Attribute ..." sowie Zugriffe auf diese "<FAMIX.Access ...". Zu den Methoden "<FAMIX.Method ..." sind deren LOC und Anzahl der Anweisungen (NOS) sowie Aufrufe der Methode "<FAMIX.Invocation ..." gegeben. Zugriffe und Aufrufe erben beide von "Association". Maße sind ebenfalls als XML-Elemente gegeben:

```
[...]
<FAMIX.Measurement xmi.id="293">
    <FAMIX.Measurement.name>
        LOC
    </FAMIX.Measurement.name>
    <FAMIX.Measurement.value>
        4
    </FAMIX.Measurement.value>
    <FAMIX.Measurement.belongsTo>
        My:ClsD.doD()
    </FAMIX.Measurement.belongsTo>
</FAMIX.Measurement>
[...]
```

In diesem Auszug beginnt das öffnende XML-Element "<FAMIX.Measurement ..." ein Maß. Das erste untergeordnete Element legt den Namen des gegebenen Maßes fest, hier "LOC". Es folgt der Wert des Maßes und eine Zuordnung zur beschriebenen Methode, hier "My:ClsD.doD()".

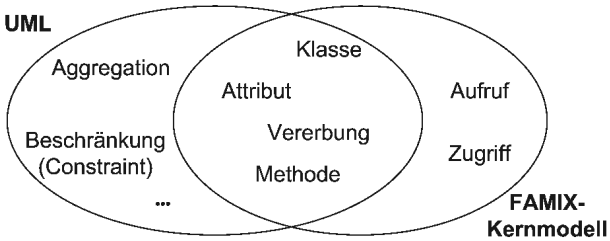


Abbildung 19: Vergleich UML – FAMIX nach [21].

Abbildung 19 zeigt exemplarisch FAMIX in Beziehung zum UML-Modell. Es gibt zwar eine gemeinsame Schnittmenge, aber kein Modell ist eine Teilmenge des anderen.

Wäre FAMIX eine Teilmenge, so könnte das Kernmodell nach [21] in UML eingebettet werden. Dort wird eine Lösung vorgeschlagen, in der den Aufrufen und Zugriffen ähnliche UML-Elemente verwendet werden. Jedoch sind komplexe Konstrukte notwendig, die die Bearbeitung verlangsamen und den benötigten Speicher erhöhen würden.

Es kann hier nur ein kleiner Teil der verfügbaren Elemente vorgestellt werden. Mehr zu FAMOOS und FAMIX ist bei [20,21,88,33] zu finden.

4.2.1.1. Statische Informationen

Statische Informationen beschreiben im Quelltext vorhandene Strukturen und Eigenschaften. Die Datenerfassung erfolgt "offline" durch die Analyse der Quell-Dateien. Typische gewonnene Informationen sind für Klassen die NOM sowie die NOA. Für Methoden liefer Analysen die LOC, lokale Variablen, gerufene Methoden u.v.m.. Aus den Informationen lassen sich Pläne erzeugen, die z.B. die innere Strukturen des Softwaresystems auf unterschiedlichen Detail-Stufen zeigen.

Problematisch ist, dass keine Aussagen zur Ausführung des Systems möglich sind. Zudem kann durch die Polymorphie nicht statisch bestimmt werden, welcher Klasse das gerufene Objekt angehört.

4.2.1.2. Dynamische Informationen

Dynamische Informationen beschreiben das Laufzeitverhalten. Die Erfassung geschieht über die Analyse von Log-Dateien, die während des Betriebs erstellt werden. Informationen die dabei für Klassen erfasst werden, sind die Anzahl der von ihren Objekten gerufenen Methoden sowie die Gesamtzahl der Methodenaufrufe. Zudem wird bestimmt, wie viele Exemplare einer Klasse erzeugt werden. Zu den Methoden werden zudem Aussagen getroffen, wie oft diese aufgerufen werden und ob dies aus der eigenen Klasse oder von außerhalb geschah.

Diese Analysen lassen kritische Punkte des Systems ("Hot-Spots") sichtbar werden. Es können z.B. oft gerufene Methoden oder oft erzeugte Objekte gefunden werden, die effizient und robust zu realisieren sind.

Dynamische Informationen haben das Problem, dass nur im Betrieb durchlaufene Situationen für die Erzeugung der Log-Daten verantwortlich sind. Es sind keine Aussagen über nicht durchlaufene Teile des Softwaresystems möglich.

CodeCrawler kann im aktuellen Stand keine dynamischen Informationen verarbeiten und beschränkt sich auf statische Eigenschaften. Deshalb werden dynamische Informationen hier nicht weiter behandelt.

4.2.2 Polymetrische Sichten

Polymetrische Sichten (engl. "Polymetric Views") sind eine Art der Informationsvisualisierung. Auf ihnen basiert die in CodeCrawler realisierte Visualisierung.

Polymetrisch ("mehrmaßig") bedeutet, dass das Ergebnis mehrere Maß-Informationen gleichzeitig zur Verfügung stellt. Die grafische Präsentation basiert auf 2D-Visualisierung, was die Anzahl der zu einem Subjekt gleichzeitig anzeigbaren Maße begrenzt.

Sicht ergibt sich aus der Notwendigkeit, die Anzahl der gleichzeitig darzustellenden Maße und Entitäten zu begrenzen. Es muss gefiltert werden. Dies ist analog zum Datenbank-Konzept der Sichten, wo ebenfalls die relevante Teilmenge der insgesamt verfügbaren Informationen gefiltert wird.

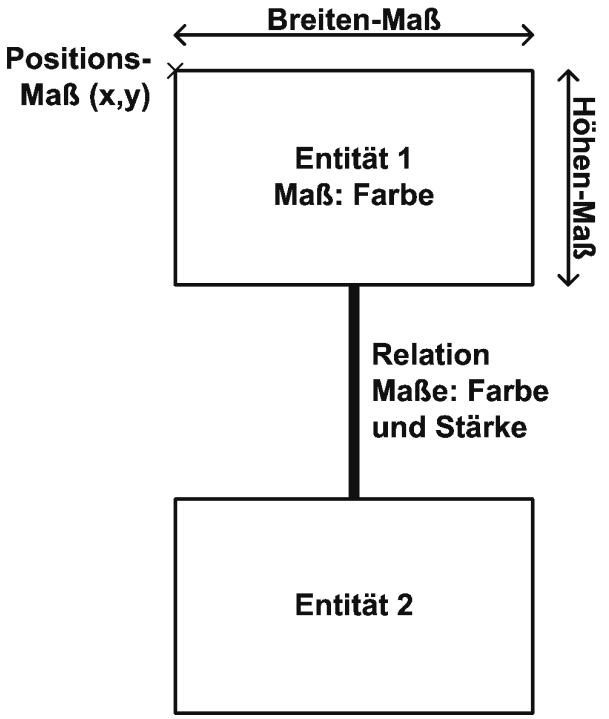


Abbildung 20: Grundlagen Polymetrischer Sichten.

Abbildung 20 zeigt das Grundkonzept polymetrischer Sichten. Rechtecke stellen Entitäten dar, Verbindungslinien Relationen. Den Bausteinen werden durch "Measurement Mapping" verfügbare Maße zugeordnet:

- Für Entitäten:
  Höhe und Breite, Farbe, x- und y-Position
- Für Relationen:
  Strichstärke und Farbe

Maße können mehrfach abgebildet werden, z.B. können Höhe und Breite das gleiche Maß visualisieren. Es ergeben sich unterschiedliche polymetrische Sichten, die nach [62] von drei Dingen abhängen:

1. Ein Layout

   definiert die Anordnung, die Abstände, die Menge der sichtbaren Knoten sowie deren Sor-

tierung. In CodeCrawler wurden unter anderem folgende Layouts realisiert.

(a) Baum

Stellt hierarchische Abhängigkeiten wie Vererbungshierarchien u.a. dar.

(b) Punktwolke (Scatterplot)

Die Knoten werden zwei Metriken entsprechend in einem orthogonalen Gitter mit Ursprung oben links angeordnet. Dieses Layout hängt von den zwei Metriken ab und nicht von der Knotenanzahl und ist geeignet, sehr viele Knoten darzustellen.

(c) Histogram

Die Knoten werden vertikal verteilt, abhängig von einer Metrik. Knoten mit gleichen Werten dieser Metrik werden nebeneinander angeordnet.

(d) Schachbrett

Die Knoten werden nach einer Metrik sortiert und in ein Schachbrett eingeordnet. Es lassen sich viele Knoten gleichzeitig darstellen und Ausreißer identifizieren.

(e) Gestapelt

Knoten werden nach ihrer Breite sortiert nebeneinandergestellt. Eine zweite Metrik wird auf die Höhe abgebildet. Unter der Annahme, die Metriken korrelieren, ergibt sich eine stetige Treppe. Außergewöhnliche Knoten werden so schnell gefunden.

2. Eine Menge von Maßen

wird für eine polymetrische Sicht aus den verfügbaren Maßen ausgewählt. Visualisierung und Interpretation des Ergebnisses hängen hiervon ab.

3. Eine Menge von Entitiäten

wird je nach Granularität der aktuellen Untersuchungsmethode ausgewählt. Die Anzahl reicht von einzelnen bis zu allen vorhandenen Entitäten.

In Abschnitt 4.3.2 werden einige polymetrische Sichten vorgestellt.

4.2.3 Analyseverfahren für objektorientierte Software

Nach der Informationsextraktion wird das Ergebnis visualisiert, was die Hauptaufgabe der Softwarevisualisierung ist. Da die Daten meistens zu umfangreich sind, müssen die dargestellten Informationen begrenzt werden, damit die Visualisierungen für den Benutzer einen Mehrwert gegenüber der Betrachtung des Quelltextes haben. Die Informationen werden auf höhere Schichten abstrahiert, bzw. die Menge der betrachteten Elemente begrenzt. So sind visuelle Muster zu erkennen, die sonst im Quelltext untergehen. Nach [62] adressieren Analysen drei Aspekte.

4.2.3.1. Coarse-grained

Grobgranulare Analysen bieten einen Einstieg in ein Softwaresystem. Es werden große Systeme betrachtet und ein Ersteindruck gewonnen, sowie näher zu untersuchende Teile identifiziert. Dies erfordert spezialisierte Verfahren, da es nicht in einem Schritt möglich ist.

First Contact Views

Diese Sichten werden verwendet, einen ersten Blick auf das System zu werfen. Lanza schlägt die folgenden Sichten vor:

- System Hotspots View
- System Complexity View
- Root Class Detection View
- Implementation Weight Distribution View

Inheritance Assessment Views

Diese Sichten unterstüzten das Verständnis der Vererbung in einem Softwaresystem. Vorgeschlagen werden:

- Inheritance Classification View
- Inheritance Carrier View
- Intermediate Abstract View

Candidate Detection Views

Sie erlauben die Identifikation von besonderen Elemente, die näher untersucht werden sollen. Dies ermöglichen:

- Data Storage Class Detection View
- Method Efficiency Correlation View
- Direct Attribute Access View
- Method Length Distribution View

4.2.3.2. Fine-grained

Feingranulare Analysen erlauben es, die innere Struktur von Klassen zu untersuchen. Die visuellen Muster (Implementations-Muster) können interpretiert werden und Optimierungsmöglichkeiten der Implemetierung gefunden werden. Es ist zudem möglich von Implementations-Mustern auf Design-Pattern [36] zu schließen.

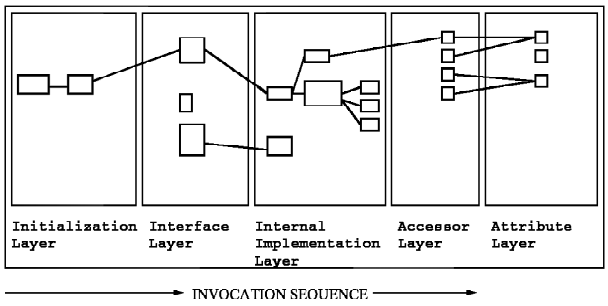


Abbildung 21: Class Blueprint Aufbau nach [62].

Class Blueprints

Diese feingranulare Analysen ordnet Elemente mit unterschiedlichen Aufgaben in der Klasse voneinander getrennt in fünf Schichten an, wie Abbildung 21 zeigt.

1. Initialization

   Diese Schicht enthält Methoden zur Initialisierung eines Objektes.
2. Interface

   Diese Methoden beschreiben das Interface der Klasse. Sie werden nur von außerhalb der Klasse oder aus der Initialisierungsschicht gerufen.
3. Internal Implementation

   Methoden, die aus anderen Methoden der Klasse (außer Initialisierungsmethoden) aufgerufen werden, sind hier eingeordnet. Sie werden nicht von außerhalb der Klasse gerufen.
4. Accessor

   Diese Methoden werden dazu verwendet, auf die Attribute der Klasse zuzugreifen.
5. Attribute

   Diese Schicht enthält in der Klasse definierte Attribute.

In [62] werden mehrere Implementations-Muster vorgestellt. Sie deuten auf Eigenschaften der Klasse hin und erlauben Kategorisierungen, wie z.B. größenbasiert (Size-based):

- Single: Klassen enthalten nur ein Knoten.
- Micro: Klassen enthalten nur wenige Methoden, z.B. wie bei spezialisierten Klassen.
- Giant: Klassen mit >>100 Methoden und Zugriffen. Eine nähere Betrachtung ist sinnvoll.
- Large Implementation: Klassen enthalten viele Knoten, oft in Sub-Schichten organisiert.

Eine andere Kategorisierung basiert auf der Verteilung in den Schichten (Layer-Distribution-based)

- Three-Layers : Gleichverteilung auf drei bis vier Schichten des Blueprints
- Interface : Schwerpunkt auf Interface-Schicht, also bei abstrakten Superklassen, oder bei "Interfaces"
- Wide-Interface: Proportional zum Rest sehr große Interface-Schicht bei vielen Zugängen zur Implementierungsschicht.

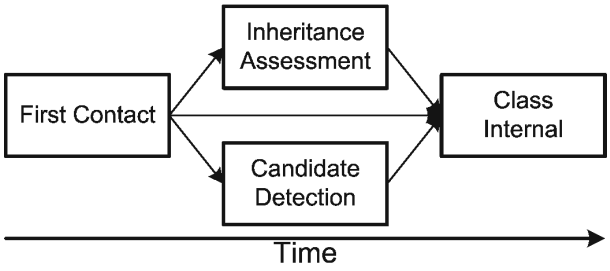


Abbildung 22: Analyseschritte nach [62].

Abbildung 22 zeigt die Reihenfolge der vorangehend vorgestellten grob- und feingranularen Analyseschritte.

4.2.3.3. Evolutionary

Evolutionäre Analysen unterstützen das Verständnis der Entwicklung von Elementen wie z.B. Klassen und deren Methodenzahl oder des gesamten Systems. So werden Muster bei der Entwicklung oder Auswirkungen neuer Versionen des Systems auf einzelne Elemente erkannt.

4.2.4 Bewertung

Durch die Analyse eines Systems von den groben zu den feinen Strukturen ist eine effiziente Einarbeitung möglich, da die gleichzeitig vom Benutzer aufzunehmenden Informationen durch Abstraktion stark verringert sind. Zudem werden nur Systemteile detailliert untersucht und die menschliche Fähigkeit der Mustererkennung dazu verwendet, Besonderheiten zu erfassen. Hierzu sind die Polymetrischen Sichten für ihre Aufgabe optimiert. Es ist auch möglich, die gewonnenen Informationen bei der Erstellung anderer Meta-Modellen zu verwenden.

Zu beachten ist jedoch, dass nicht alles 1:1 abgebildet werden kann und somit teilweise komplexe Konstruktionen notwendig sind, die die Verständlichkeit der Informationen verringern.

4.3 Praktische Untersuchung von CodeCrawler

Im Folgenden wird anhand einer beispielhaften Untersuchung von DoxyGen ein kurzer Ablauf einer Softwareanalyse vorgestellt. DoxyGen ist ein in C++ geschriebenes System zur automatisierten

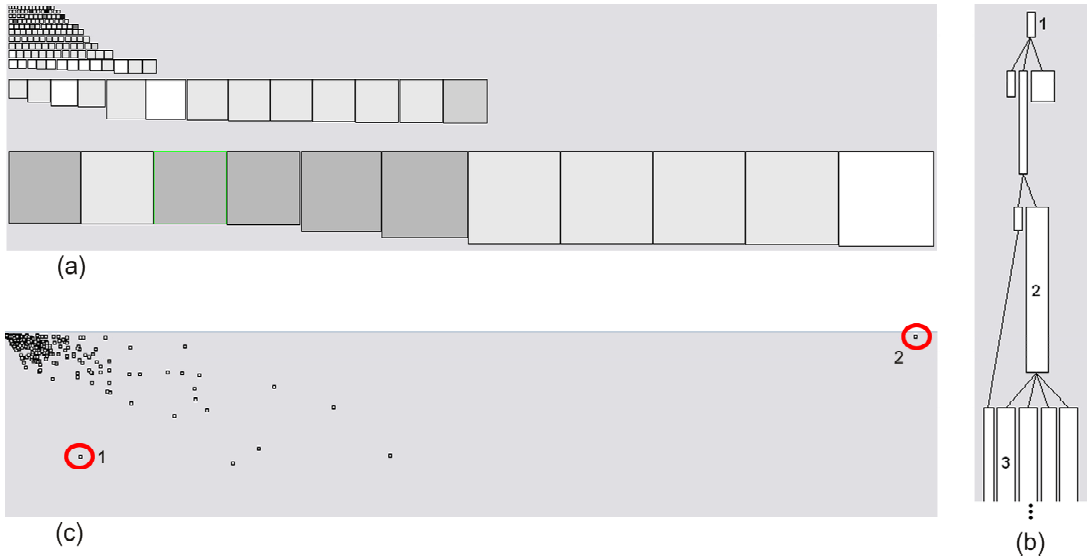


Abbildung 24: (a)System Hotspot Sicht, (b) System Komplexitäts Sicht und (c) Methoden Effizienz Korrelations Sicht.

Erzeugung von Quelltext-Dokumentationen mit unterschiedlichen Ausgabeformaten.

4.3.1 Notwendige Werkzeugumgebung

Neben CodeCrawler zur Visualisierung der Software sind noch andere Werkzeuge notwendig, um die Quelldateien hierfür vorzubereiten.

4.3.1.1. Vorbereitung der Daten

Für C++-Systeme bietet das Werkzeug Columbus/CAN [35] die Vorverarbeitung der Quelldateien. Abbildung 23 zeigt den Ablauf von den Quell- bis zu den FAMIX-Dateien, die in CodeCrawler eingelesen werden. Nachdem ein C++-Präprozessor die Quellen vorbereitet hat, verarbeitet ein C++-ANalyzer (CAN) dieses Zwischenformat zu CAN Symboltabellen. Ein CAN-Linker fügt diese in einem abstrakten Syntax Graphen (ASG) zusammen und speichert sie als verbundene (merged) Symboltabelle. Die relevanten Teile dessen werden ausgewählt und ins FAMIX-Format exportiert.

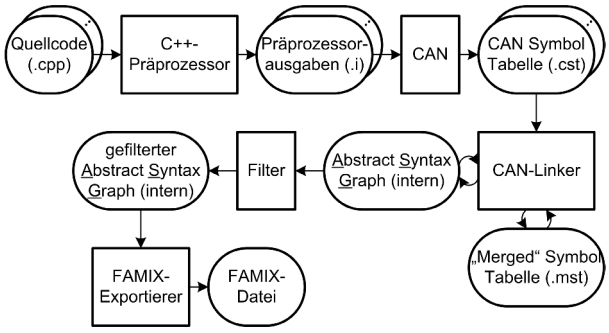


Abbildung 23: Columbus/CAN Analyseablauf und Export.

Nur wenn die Quelldateien verändert wurden ist ein erneutes Vorverarbeiten notwendig, ansonsten kann die verbundene Tabelle erneut geladen werden. Der Export ist dann sofort möglich.

4.3.1.2. Laden der vorbereiteten Daten

Die erzeugte FAMIX-Datei wird von CodeCrawler geladen. Dies geschieht durch MOOSE, eine in CodeCrawler integrierte SmallTalk-Implementierung des FAMIX-Models.

4.3.1.3. Anpassen der Sichten

In CodeCrawler sind bereits Sichten vordefiniert, die mit dem integrierten Sichten-Editor (View Editor) anpassbar sind. Es können weitere Sichten angelegt werden. Die relevanten Entitäten und Metriken werden ausgewählt und den grafischen Elemente zugeordnet.

4.3.2 Leistungsmerkmale des untersuchten Werkzeugs

Nachfolgend sind anhand der Untersuchung von DoxyGen einige polymetrische Sichten gegeben.

4.3.2.1. System Hotspot (First Contact)

Diese Sicht erlaubt es, besonders große oder kleine Klassen zu identifizieren und skaliert bis zu sehr großen Systemen. Es werden Metriken wie NOM oder NOA für die Breite und Höhe der Knoten und die Tiefe in der Vererbungshierarchie (Hierarchy Nesting Level, HNL) für die Farbe verwendet. Die Knoten werden in einem Schachbrett angeordnet.

Abbildung 24 (a) zeigt die beschriebene Sicht auf die Klassen von DoxyGen. Es werden jeweils gleich viele Klassen je Zeile angezeigt, die Knotenhöhe und -breite resultiert aus der Anzahl der Methoden, wonach auch sortiert wird. So können

auffällige Knoten identifiziert werden. Dass diese Sicht für große Systeme skaliert ist anzunehmen, da je hinzukommender Klasse nur ein grafisches Element erzeugt wird.

4.3.2.2. System Complexity (First Contact)

Nachdem ein erster Eindruck der Größe des Systems und einzelner Klassen gewonnen wurde, werden die Vererbungshierarchien untersucht. Dies gibt Hinweise auf die Komplexität. Vererbungshierarchien können durch die Verfahren aus Abschnitt 4.2.3.1 näher analysiert werden.

Die Darstellung erfolgt in einem Baum-Layout, in dem die Klassen über die Vererbungsbeziehungen miteinander verbunden sind. Die Breite der Klassenknoten visualisiert die NOA, die Höhe die NOM. Die Farbe ergibt sich aus der Gesamtzahl der LOC über alle Methoden (WLOC).

Bei großen untersuchten Systemen empfiehlt sich die Analyse von Teilsystemen. Abbildung 24 (b) zeigt die Vererbungshierarchie der Ausgabe-Generatoren. Hier ist zu erkennen, dass es eine Klasse BaseCodeDocInterface (1) gibt, von der alle Ausgabeklassen erben. Unter anderem ist dies die Klasse OutputGenerator (2), von der spezialisierte Ausgabeklassen wie z.B. LatexGenerator (3) erben.

Solche hierarchischen Strukturen deuten auf Spezialisierung der Aufgaben in tieferen Hierarchiestufen hin, wie z.B. auch in GUI-Klassenhierarchien.

4.3.2.3. Method Efficiency Correlation (Candidate Detection)

Diese Sicht ordnet alle Methoden mittels Punktwolken-Layout an, mit Ursprung oben links. Hierfür werden LOC für die x-Position und die Anzahl der Anweisungen (NOS) für die y-Position verwendet. Die Methoden sind entlang der Diagonalen von oben links nach unten rechts zu erwarten. Diese Sicht erlaubt es, leere, tote[3], überlange oder übermäßig komplexe[4] Methoden zu identifizieren.

In Abbildung 24 (c) ist ein Bespiel dieser Sicht zu erkennen. Methoden, die weit oben links sind werden als leer, diejenigen weit oben rechts (z.B. 2) als tot und diejenigen weit unten als lang bezeichnet. Methoden unten links (z.B. 1) sind übermäßig komplex und sollten näher untersucht werden. Die Analyse des Quelltextes von Methode 1 ergibt, dass überwiegend "switch(...) case ..." mit mehreren Anweisungen je Zeile vorkommen, z.B. "case DocSymbol::Less: m_t << "<"; break;". Methode 2 enthält entgegen der Erwartung nur wenig Kommentare, jedoch werden in den Anweisungen Zeichenketten von mehreren Zeilen Länge verarbeitet, was das hohe LOC-Maß erklärt.

4.3.2.4. Class Blueprint (fine-grained)

Nachdem der Systemüberblick gewonnen wurde, werden einzelne Klassen näher untersucht. Für die Methoden werden die Anzahl der Aufrufe (No of Invocations, NI), für die Breite und die LOC für die Höhe verwendet. Abstrakte Methoden sind in CodeCrawler Cyan-farbig.

Die Attribute werden mit Zugriffe von innerhalb der Klasse (NLA) für die Breite und Zugriffe von außen (NGA) für die Höhe verwendet.

Abbildung 25 zeigt die Klasse LatexGenerator einmal als Ganzes (1) und einmal einen Teil (2). Es sind implementierte Methoden und Zugriffe aus Methoden auf Attribute sichtbar.

Zudem wird die Basisklasse aus Abschnitt 4.3.2.2 dargestellt, BaseCodeDocInterface (3). Nur die Interface-Schicht enthält Methoden, was dem Interface-Muster entspricht.

Die Analyse von C++-Software ergibt, dass die Konstruktoren nicht in der Initialisierungsschicht sind, da sie den Klassennamen enthalten und nicht "init" oder "initialize", wie von CodeCrawler benötigt.

Werden mehrere Klassen gleichzeitig als Blueprints angezeigt, so werden zudem die Zugriffe auf Methoden und Attribute anderer Klasse dargestellt.

4.3.3 Bewertung

CodeCrawler hilft dem Benutzer bei der Einarbeitung in ein System, indem Implementationsdetails durch grobgranulare Sichten stark abstrahiert werden. Durch Analyseabfolgen lassen sich hiervon ausgehend die Teile des Systems identifizieren, die näher zu betrachten sind.

Die Darstellung der Daten ist zwar geeignet, die Navigation geschieht jedoch sprunghaft. Somit ist die Orientierung schnell verloren, was durch veränderte Knotenanordnungen verschlimmert wird.

Zudem muss für die effiziente Verwendung die VisualWorks SmallTalk Entwicklungsumgebung verwendet werden, da sonst fehlende Fehlerbehandlung die Analysen erschweren.

Positiv ist die Möglichkeit, mehrere Sichten auf Daten in unterschiedlichen Fenstern parallel zu haben. Es können dabei die gleichen Elemente in unterschiedlichen Sichten, ebenso wie unterschiedliche Elemente durch gleiche Sichten gezeigt werden. Dies bietet gute Vergleichsmöglichkeiten und minimiert die notwendige Navigation.

[3] Die Methoden enthalten nur Kommentare.
[4] Die Methoden enthalten mehrere Anweisungen je Zeile.

Abbildung 25: Class Blueprint und ein Teilauszug aus der Klasse LatexGenerator sowie der "Klasse" BaseCodeDocInterface.

Durch die Analyse von Implementations-Mustern werden zudem Systemeigenschaften sichtbar, die zur Optimierung verwendet werden können, z.B. durch Anwendung von Design-Pattern.

4.4 Einordnung und Bewertung

4.4.1 Abbildungsmächtigkeit in Bezug auf UML-Systemmodelle

- Abbildung der Elemente der statischen Systemarchitektur

  Die verfügbaren Elemente werden durch das Meta-Format FAMIX festgelegt. Das Modell [20] erlaubt Pakete, Klassen, Methoden, Attribute, mehrere Variablenarten, spezifizierte Assoziationen wie Zugriff oder Aufruf. Wie Abbildung 19 zeigt, ist die Aggregation nicht in FAMIX integriert, was aber die Abbildung von FAMIX nach UML nicht verhindert. FAMIX-Elemente, die UML nicht enthält, können durch Kombination mehrerer UML-Elemente ersetzt werden. Allerdings sind die Ergebnisse mit hohem Aufwand verbunden und nicht 1:1 abbildbar [21].

- Abbildung der Elemente der dynamischen Systemarchitektur

  Momentan kann CodeCrawler keine dynamischen Eigenschaften visualisieren. Es gibt jedoch erste Ansätze [25].

4.4.2 Klassifikation des Software-Visualisierungsansatzes

- Visualisierungsprinzip

  Entitäten werden als Rechtecke und Beziehungen als Verbindungslinien dargestellt. Dieses Prinzip wird durch polymetrische Sichten erweitert. In [62] werden mögliche Interpretationen für visuelle Muster beschrieben, die bei der Analyse auftreten.

- Visualisierungsprozess

  Die Programmquellen werden zunächst vorverarbeitet und im FAMIX-Format abgelegt. Dies wird in CodeCrawler geladen und zur Visualisierung verwendet.

- Intention des Ansatzes

  Der Ansatz ist im Rahmen des FAMOOS-Projektes zum Reengineering von objektorientierter Software entstanden. Er dient der Erst- und weitergehenden Analyse und soll frühzeitig verwendbare Ergebnisse liefern.

- Unterstützte Programmiersprachen

  Nach [33] werden für das FAMIX Modell C++, Java, Ada und Smalltalk unterstützt. Es

ist jede Sprache denkbar, für die ein Extrahierer nach FAMIX existiert.

- Programmierkonventionen

  Es müssen keine speziellen Elemente zu den Programmquellen hinzugefügt werden, jedoch sind z.B. für Class Blueprints Konventionen einzuhalten, damit die Methoden in den richtigen Schichten eingeordnet werden. Initialisierungsmethoden müssen z.B. "init" enthalten, was bei C++- oder Java-Konstruktoren nicht möglich ist.

- Skalierbarkeit bezogen auf die Systemgröße

  Unter anderem wurden folgende Systemgrößen untersucht:

Sprache	LOC	Klassenzahl
C++	1.2M	>2300
C++/Java	120k	>400
Smalltalk	600k	>2500

- Automatisierungsgrad

  Durch vordefinierte Sichten ist für eine Erstanalyse keine Konfiguration notwendig. Die Erstellung der FAMIX-Daten im Vorverarbeitungsschritt erfordert jedoch eine genaue Präprozessorkonfiguration.

4.4.3 Werkzeugeigenschaften

CodeCrawler kann in die VisualWorks SmallTalk Entwicklungsgebung geladen werden und SmallTalk-Systeme analysieren. Für andere Sprachen gibt es keine Integration in eine Entwicklungsumgebung. Es müssen dann FAMIX-Daten erstellt werden, um diese einzulesen.

CodeCrawler ist als "parcel" für SmallTalk und als Windows-Executable frei verfügbar auf [88]. Es wird so jede Plattform unterstützt, für die eine SmallTalk Laufzeitumgebung existiert. Für Fehlertoleranz ist die SmallTalk Entwicklungsumgebung notwendig.

4.4.4 Forschung und Entwicklung

- Informatikfakultät (Uni Lugano)
- Prof. Dr. Michele Lanza
- Institut für Informatik und angewandte Mathematik (Uni Bern)
- Prof. Dr. Oscar Nierstrasz und (Assistenz) Prof. Dr. Stéphane Ducasse
- Beim FAMOOS-Projekt auch Daimler-Benz, Nokia, SEMA Spain und Take5

4.5 Schlussfolgerungen

CodeCrawler unterstützt den Benutzer bei der Erstanalyse eines Systems, indem es einen Eindruck von Größe und Komplexität bietet. Es hilft Teile des Systems zu identifizieren, die näher zu untersuchen sind, was sowohl ungenutze als auch zu optimierende Stellen sein können. Die Analyse selbst wird durch viel theoretisches Material zum Werkzeug und den realisierten Visualisierungen unterstützt. Unzusammenhängende Dokumentation lässt die Einarbeitung in die Werkzeugumgebung jedoch "experimentell" wirken.

Durch die Vererbungshierarchien und Kollaborationsbeziehungen unterstützt CodeCrawler bei der Erarbeitung von Architekturplänen. Implementations-Muster lassen zudem Schlüsse auf verwendete Design-Pattern zu. Ebenso geben sie Hinweise auf Stellen, die durch Design-Patterns verbessert werden können.

CodeCrawler und die realisierten Visualisierungsmechanismen objektorientierter Softwaresysteme unterstützen bei der Einarbeitung in ein System. Die Informationen können bei der Erstellung von Dokumentationen und Plänen verwendet werden, da bekannte Elemente, z.B. zum Systemdesign, wiederzufinden sind. Es sind auch Kombinationen mit anderen Meta-Modellen möglich. Da jedes Meta-Modell für eine besondere Aufgabe optimiert ist und diese Unterschiede nicht immer trivial zu überbrücken sind.

Kapitel 5

CodeCrawler in der Anwendung

Benjamin Hagedorn

Kapitel 5: CodeCrawler in der Anwendung

Benjamin Hagedorn

Zusammenfassung. CodeCrawler ist ein Werkzeug zur zweidimensionalen Visualisierung von Software. Dieser Artikel gibt einen kurzen Überblick über die Nutzbarkeit von CodeCrawler für die Analyse eines unbekannten Softwaresystems. Zielsystem ist das FTP-Programm lftp. Wir zeigen welche Vorbereitungen für die Erstellung einer Visualisierung notwendig sind und welche Ansichten von CodeCrawler für die Erstanalyse eines Softwaresystems geeignet sind.

5.1 Einführung

Softwaresysteme besitzen oft eine hohe Komplexität. Aus dem Quelltext des Softwaresystems und anderen Quellen kann eine Vielzahl von Softwareinformationen entnommen und hergeleitet werden. Verschiedene Nutzer benötigen für verschiedene Aufgaben unterschiedliche Teilmengen dieser Informationen. Die Visualisierung der Softwareinformationen kann den Benutzer bei der Wahrnehmung und der Verständnisgewinnung über das Softwaresystem unterstützen. Softwarevisualisierungssysteme versuchen deshalb eine Auswahl der Softwareinformationen auf geeignete Repräsentationen abzubilden.

Es existieren Ansätze sowohl zur zweidimensionalen als auch zur dreidimensionalen Softwarevisualisierung.

5.2 Konzept von CodeCrawler

CodeCrawler ist ein von Michele Lanza entwickeltes Werkzeug zur zweidimensionalen Visualisierung von objektorientierter Software und soll der Unterstützung des Reverse-Engineerings dienen. Lanza unterteilt den Prozess des Reverse-Engineerings in vier große Aufgabenbereiche (nach [62]):

- Erstkontakt mit dem System
- Bewertung von Vererbungsstrukturen
- Ermittlung von interessanten Kandidaten
- Genaue Untersuchung von Klassen

5.2.1 Polymetrische Sichten

CodeCrawler visualisiert die jeweils benötigten Softwareinformationen in einer Reihe von Sichten: Grundelemente einer jeden Visualisierung in CodeCrawler sind Rechtecke verschiedener Größe, Färbung und Position. Diese Rechtecke bilden Informationsentitäten ab. Beziehungen zwischen Entitäten werden durch verschieden starke und gefärbte Linien zwischen den Rechtecken dargestellt. Sichten können auch durch den Benutzer erzeugt und verändert und damit dem Bedarf angepasst werden.

CodeCrawler unterstützt drei Typen solcher so genannten Polymetrischen Sichten:

- Grobgranulare Sichten dienen dazu, einen ersten Eindruck vom System zu erhalten.
- Feingranulare Sichten dienen der genauen Untersuchung einzelner Strukturen.
- Evolutionäre Sichten dienen der Visualisierung von Softwareprozessinformationen.

5.2.2 Erstanalyse eines Systems

Für die Erstanalyse eines Softwaresystems sind im Besonderen die grobgranularen Sichten von Interesse. Nach Lanza können diese im Rahmen des Reverse-Engineerings bei einer Reihe von Aufgaben hilfreich sein:

- Bewertung der allgemeinen Qualität des Softwaresystems
- Übersichtsgewinnung in Bezug auf Größe, Komplexität und Struktur
- Lokalisierung der wichtigsten Klassen und Vererbungshierarchien
- Identifizierung von außergewöhnlich großen oder komplexen Klassen zur genaueren Untersuchung
- Lokalisierung von ungenutztem Code

Analog zu den anfangs identifizierten Aufgabenbereichen unterteilt Lanza die Sichten für den Erstkontakt noch einmal in:

- First Contact Views
- Inheritance Assessment Views
- Candidate Detection Views

5.3 Einsatz von CodeCrawler

5.3.1 lftp

Als zu untersuchendes Softwaresystem soll lftp [63] dienen. Das Programm ist ein Client für den Dateitransfer. Es wurde von Alexander V. Lukyanov entwickelt und ist unter der GNU GPL frei verfügbar.

Das Programm lftp arbeitet kommandozeilenorientiert. Es unterstützt eine Vielzahl von Übertragungsprotokollen (FTP, HTTP, FISH, SFTP, HTTPS, FTPS). Das Besondere an lftp ist die zuverlässige Datenübertragung, die so oft wiederholt wird, bis die Daten übertragen wurden. Es unterstützt zudem die Datenspiegelung zwischen Client und Server. Lftp besitzt eine Auftragsverwaltung: Aufträge können in eine Warteschlange eingestellt und auch gelöscht werden. Aufträge können auch im Hintergrund gestartet werden.

Das Programm ist in der Programmiersprache C++ entwickelt worden. Unter Windows ist lftp in der Cygwin-Umgebung ausführbar. Das Hauptprogramm umfasst etwa 2,5 MB Daten, die sich auf 278 Dateien mit 110.615 Quelltextzeilen (LOC) und 228 Klassen verteilen.

5.3.2 Vorbereitungen

CodeCrawler basiert auf Moose, einer Implementierung des FAMIX-Metamodells [24,33].

FAMIX ist eine XML-Sprache zur Beschreibung von Informationen über ein Softwaresystem. CodeCrawler kann sowohl CDIF- als auch XMI Dateien im FAMIX-Format verarbeiten. Die FAMIX-Beschreibung muss aus dem Quelltext von lftp abgeleitet werden. Dazu haben wir das Werkzeug Columbus/CAN verwendet [35]. Um auch Dateien im älteren FAMIX-Format 2.0 unterstützen zu können muss das Ergebnis mit einem Shellscript angepasst werden.

CodeCrawler selbst ist in Smalltalk entwickelt. Bei der Ausführung des Programms treten an vielen Stellen unbehandelte Ausnahmen auf. Bei der Verwendung der Windowsversion von CodeCrawler führt dies zum sofortigen Abbruch des Programms. Der Programmabbruch kann durch die Verwendung der Smalltalk-Pakete und durch die Benutzung einer Smalltalk-Laufzeitumgebung zum Abwickeln von CodeCrawler verhindert werden. Dies ist auch deshalb sinnvoll, da gerade das Laden der FAMIX-Modelle bei sehr großen zu visualisierenden Programmen sehr lange dauert.

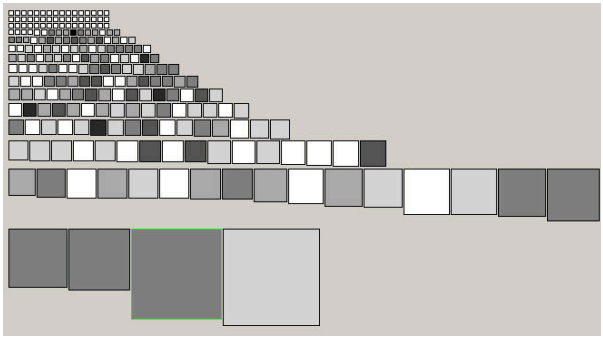


Abbildung 26: System Hotspot View von lftp.

5.3.3 Analyse

Für die grobe Analyse von lftp sind im Besonderen die grobgranularen Sichten von CodeCrawler hilfreich. Im Folgenden werden wir die verwendeten Sichten am Beispiel beschreiben und kurz die Erkenntnisse für den Verständnisprozess erläutern.

5.3.3.1. First Contact Views

First Contact Views dienen der Gewinnung eines ersten groben Eindrucks von einem Softwaresystem. Es geht darum die Klassen kennen zu lernen und erste Vermutungen über ihren funktionalen Umfang und ihre Komplexität abzuleiten.

System Hotspot View

Die System Hotspot View visualisiert jede Klasse durch ein Rechteck. Ihre Breite und Höhe repräsentieren die Anzahl der enthaltenen Methoden (NOM) der Klasse. Die Färbung der Rechtecke weist zusätzlich auf die Tiefe dieser Klasse im Vererbungsbaum (HNL) hin.

Abbildung 26 zeigt die System Hot Spot View für lftp. Es wird deutlich, dass lftp 228 Klassen besitzt. Die Klassen mit der größten Zahl an Methoden sind *FileAccess*, *Ftp*, *CmdExec*, *SFtp*, *Http*, *Fish* und *Job*. Mit Hilfe der Dateinamen können wir vermuten, dass diese Klassen die Hauptfunktionalität enthalten – dies sind der Zugriff auf die zu übertragenden Dateien, die verschiedenen Protokoll-Handler und Teile der Auftragsverwaltung. Dies sind also Klassen, für die eine genauere Untersuchung sinnvoll wäre. Die Tiefe im Vererbungsbaum weist zudem auf eine eventuell große Komplexität hin.

Auf der anderen Seite werden auch solche Klassen identifiziert, die nur sehr wenige oder keine Methoden enthalten.

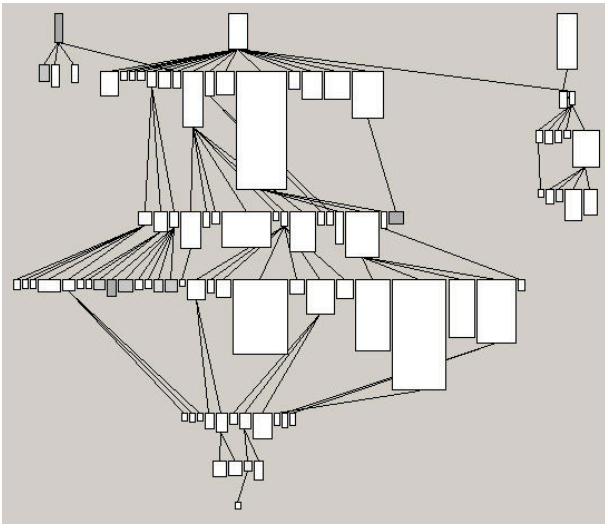


Abbildung 27: Ausschnitt aus der System Complexity View von lftp.

System Complexity View

Die System Complexity View visualisiert die Vererbungsstrukturen im Programm. Abbildung 27 zeigt einen Ausschnitt aus der System Complexity View von lftp. Die Rechtecke repräsentieren Klassen, die Verbindungen stellen die Vererbungsbeziehungen dar. Die Höhe der Rechtecke repräsentiert wiederum die NOM jeder Klasse. Ihre Breite ist ein Maß für die Anzahl der Attribute der Klasse (NOA). Hier wird ersichtlich, dass die Protokoll-Klassen im rechten unteren Teil des Vererbungsbaumes alle von der Klasse *NetAccess* abgeleitet sind, die wiederum von *FileAccess* erbt und vermutlich gemeinsame Funktionalität zur Verfügung stellt. Wegen der Vererbungstiefe und möglicher Polymorphie sind an dieser Stelle komplexe Strukturen und komplizierte Ausführungspfade zu erwarten. Die Protokoll-Klassen *FtpS*, *HFtp* und *Https* sind als Ableitung von *Ftp* bzw. *Http* realisiert. Ein weiterer tiefer Ast geht von der Klasse *Job* aus. Abgeleitet sind zum Beispiel *SessionJob* oder *MirrorJob*. Nicht dargestellt sind eine Reihe von kleinen Klassen, die lose und ohne tiefe Vererbungsstrukturen existieren.

5.3.3.2. Inheritance Assessment Views

Diese Gruppe von Sichten erlaubt Aussagen über die Verwendung von Vererbung innerhalb des untersuchten Systems. Beispiele sind die Inheritance Classification View und die Inheritance Carrier View.

Inheritance Classification View

In der Inheritance Classification View bilden Rechtecke wieder die Klassen ab, ihre Höhe gibt die Anzahl der überschriebenen Methoden (NMO) an, die Breite die Anzahl der hinzugefügten Methoden (NMA). Die Farbe weist auf die Anzahl erweiterter Methoden (NME) hin. Diese Sicht klassifiziert also die Vererbungsbeziehungen.

Abbildung 28 zeigt einen Ausschnitt aus der Inheritance Classification View für lftp. Wir können erkennen, dass keine Klasse geerbte Methoden erweitert und maximal 7 bis 9 geerbte Methoden überschrieben werden. Einzelne Klassen (*Job*, *FileAccess*, *CmdExec*, *SFtp*, *Ftp*, *Fish*, *Http*) erweitern die geerbte Funktionalität durch Hinzufügen von vielen Methoden.

Inheritance Carrier View

Die Höhe und Breite der Rechtecke in der Inheritance Carrier View bilden die Anzahl von Kindklassen (WNOC) und die NOM einer Klasse ab und weisen damit auf die Bedeutung der Klasse für ihre Subklassen hin. Die Visualisierung von lftp weist auf die bisher nur wenig beachtete Klasse *SMTask* hin, die Ausgangsknoten der Vererbungshierarchie ist. Sie hat jedoch eine große Zahl an Kindern und mit ihren 27 Methoden einen großen Einfluss auf das Systemverhalten.

5.3.3.3. Candidate Detection Views

Candidate Detection Views helfen solche Softwareelemente zu identifizieren, die einer genaueren Betrachtung unterzogen werden sollten. CodeCrawler bietet dazu zum Beispiel die Method Efficiency Correlation View und die Direct Attribute Access View an.

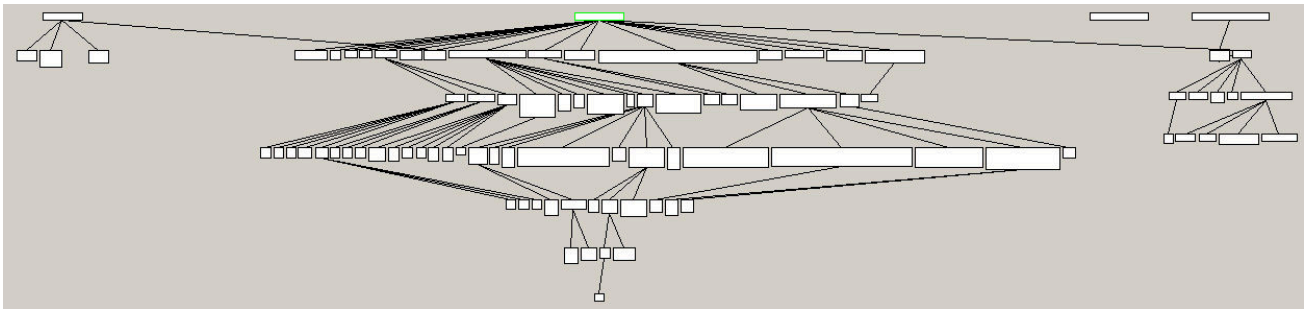


Abbildung 28: Ausschnitt aus der Inheritance Classification View von lftp.

Method Efficiency Correlation View

In der Method Efficiency Correlation View repräsentieren gleich große Rechtecke die Methoden des Systems. Diese werden entsprechend ihrer LOC und ihrer Anzahl an Anweisungen (NOS) platziert. So sind leicht solche Methoden zu identifizieren, die sehr viele Anweisungen und potentiell sehr viel Funktionalität enthalten. Außerdem wird das Verhältnis von Quelltextzeilen einer Methode und tatsächlich enthaltenen Anweisungen, und damit auch ein Aspekt der Softwarequalität verdeutlicht: Verhältnismäßig viele Anweisungen deuten auf komplexen Quelltext hin – verhältnismäßig wenige Anweisungen deuten auf sehr viele Kommentare oder sogar so genannte „tote Klassen“ ohne funktionalen Quelltext hin.

Bei lftp fällt auf, dass die Methode *Do( )* der Klasse *Ftp* mit 1224 LOC und 435 NOS auffällig umfangreich ist. Auch die Methoden *Do( )* der anderen Protokoll-Klassen sind verhältnismäßig groß. Es ist zu vermuten, dass diese die Hauptfunktionalität des Programms enthalten. Im Rahmen von Reverse-Engineering könnten diese Methoden vielleicht refakturiert werden.

Direct Attribute Access View

Die Direct Attribute Access View bildet die Attribute des Systems als Rechtecke ab. Deren Größe und Farbe werden durch die Anzahl der direkten Zugriffe auf dieses Attribut (NAA) bestimmt. Dies kann auf unbenutzte Attribute hinweisen oder auch auf sehr häufige Zugriffe, die nicht über Zugriffsmethoden erfolgen. Auch lftp weist sowohl unbenutzte Attribute als auch eine Reihe sehr oft zugegriffener Attribute auf. Die häufigsten Zugriffe erfolgen auf das Attribut *conn* der Klasse *Ftp*.

5.4 Zusammenfassung

Anfängliche Probleme beim Erzeugen der FAMIX-Repräsentation und beim Importieren ließen sich durch die Deaktivierung von Includes beim Parsen des Quelltextes und die Begrenzung des Exports von Columbus/CAN lösen.

CodeCrawler bietet eine Vielzahl von Möglichkeiten für die Analyse eines Softwaresystems. Es sind bereits sehr viele Sichten vordefiniert. Der Benutzer kann diese leicht anpassen oder neue Sichten erstellen. Die Benutzerführung von CodeCrawler ist teilweise jedoch noch fehlerbehaftet und eher arbeitsbehindernd.

Es gibt Fehler und Inkonsistenzen bei der Darstellung und Fehler bei der Programmausführung. Der Aspekt der Navigation innerhalb der Sichten ist bisher nicht ausreichend berücksichtigt.

Mit Hilfe der beschriebenen grobgranularen Sichten haben wir einen Eindruck über die Größe und potentielle funktionale Bedeutung der verschiedenen Klassen und Methoden gewonnen. Außerdem haben wir damit Anhaltspunkte für eine genauere Betrachtung der Klassen, Methoden und Attribute erhalten. Zur weiteren Analyse bietet CodeCrawler eine Reihe feingranularer Sichten oder aber den direkten Zugriff auf den Quelltext an.

Die Arbeit mit CodeCrawler erfordert das Wissen um die darstellbaren Metriken und deren Bedeutung. Mit ein wenig Übung kann CodeCrawler unserer Meinung nach gut eingesetzt werden, um die grobe Struktur eines Softwaresystems zu erfassen.

Abkürzungsverzeichnis

CASE	Computer Aided Software Engineering
CDIF	CASE Data Interchange Format
FAMIX	FAMOOS Information Exchange Model
FAMOOS	Framework-based Approach for Mastering Object-Oriented Software Evolution
FISH	File transfer with a Shell
FTP	File Transfer Protocol
FTPS	File Transfer Protocol over SSL Secure Sockets Layer
GNU GPL	Gnu General Public License
HNL	Hierarchy Nesting Level
HTTP	Hypertext Transfer Protocol
LOC	Lines of Code
NAA	Number of Attribute Access
NMA	Number of Methods Added
NME	Number of Methods Extended
NMO	Number of Methods Overridden
NOA	Number of Attributes
NOM	Number of Methods
NOS	Number of Statements
SFTP	Simple File Transfer Protocol
SHTTP	Secure Hypertext Transfer Protocol
WNOC	Number of Children
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Kapitel 6

Quellcodezeilenbasierte Softwarevisualisierung

Nebojsa Lazic

Kapitel 6: Quellcodezeilenbasierte Softwarevisualisierung

Nebojsa Lazic

Zusammenfassung. *Das vorliegende Paper beschäftigt sich mit „Codezeilen-basierter Softwarevisualisierung" (CZB SV). Ausgehend vom ersten CZB SV-Tool (SeeSoft, 1992) der Gruppe um Stephen G. Eick von den Bell-Labs werden die Konzepte, die Visualisierungstechnicken und die Benutzbarkeit dieser Art von Softwarevisualisierungssystemen vorgestellt; dabei werden auch aktuelle(re) CZB SV-Tools (Tarantula/ GAMMATELLA, Augur) verschiedener anderer Gruppen betrachtet, wobei auch der Schritt in die 3D CZB SV (sv3D) berücksichtigt wird.*

6.1 Einführung

Stasko et al [96] definieren Softwarevisualisierung wie folgt:

„Software Visualization is [..] the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software."

Softwarevisualisierung ist als Unterkategorie der Informationsvisualisierung (Visualisierung von Daten jeglicher Art) zu sehen. Informationen zu visualisieren bringt Vorteile:

- höherer Informationsgehalt bzw. höhere Informationsdichte
- höherer Abstraktionsgrad und damit bessere Abbildung des Problembereichs
- bereitstellen eines Überblicks, Strukturen werden leichter sichtbar
- besser zugänglich, leichter zu verstehen, schneller zu begreifen, leichter einzuprägen

Es stellt sich die Frage, warum die Softwareindustrie Visualisierung braucht? Mit zunehmendem Fortschritt in verschiedensten Bereichen der Informatik, ist es Softwareingenieuren möglich, immer komplexere Softwaresysteme zu entwickeln. Das ist auch mit Nachteilen verbunden:

„The fundamental cause of the software crisis is that massive, software-intensive systems have become unmanageably complex."
(Grady Booch 1994)

Damit würde auch die Hauptaufgabe, nämlich die Erweiterung und Wartung („Software Maintenance") bereits bestehender Softwaresysteme, immer schwieriger.

Die Softwarevisualisierung spielt eine wichtige Rolle im Verstehensprozess eines Softwaresystems, welcher nötig ist, um effektiv ein Softwaresystem handhaben zu können (Diagramm 1).

Sie bildet eine Erweiterung des Verstehens-

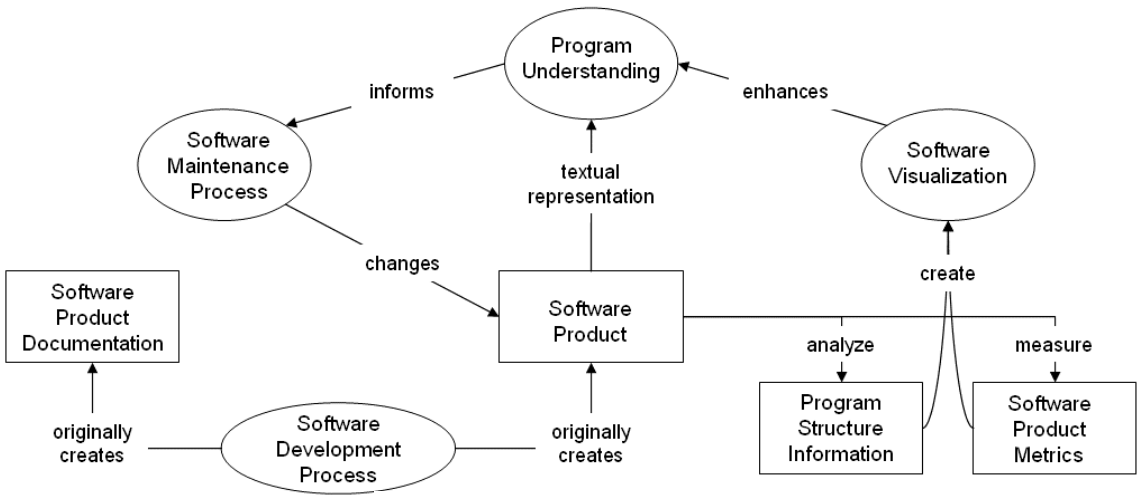


Diagramm 1, [108]
Die Rolle der Softwarevisualisierung.

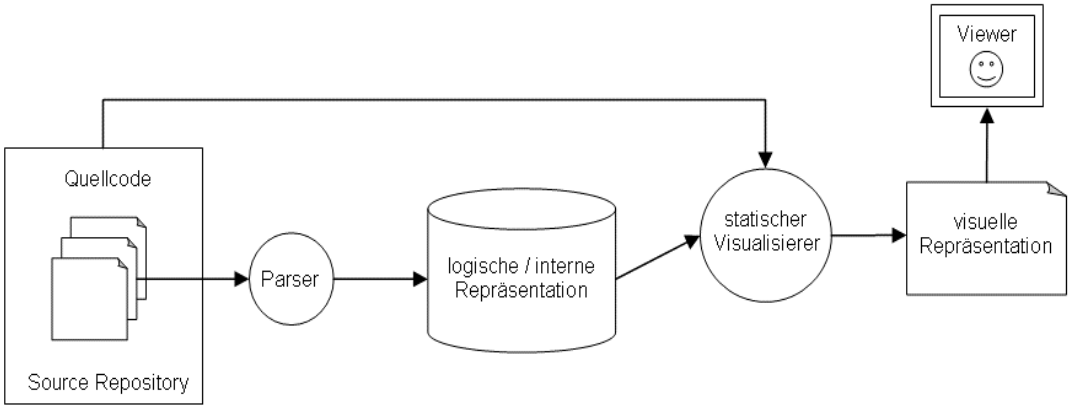


Diagramm 2, [110]
Statische Softwarevisualisierung.

prozesses, indem sie auf dem Softwareprodukt aufsetzend zum einen über „Analysen“ Programmstrukturinformationen bzw. zum anderen über „Messungen“ sowohl im statischen als auch dynamischen Bereich Produktmetriken bereitstellt.

Der Softwareingenieur erhält neben der sonst überwiegend in textueller Form vorliegenden Repräsentation des Softwaresystems, eine reichhaltigere, in der Regel komprimiertere visuelle Darstellung relevanter Informationen des zugrundeliegenden Softwaresystems.

Das Hauptziel ist, kurzgefasst: Zeiteinsparungen führen zu Kosteneinsparungen.

Die ohnehin finanziell gebeutelte Softwareindustrie verlangt nach Mitteln und Wegen, den Softwareentwicklungsprozess in allen seinen Facetten beherrschen zu können.

Grundlegend wird in der Softwarevisualisierung zwischen statischer und dynamischer Visualisierung unterschieden.

Die im vorliegenden Paper berücksichtigten Tools visualisieren nicht explizit das sich in der Ausführung befindliche Softwaresystem (ausgegraute Elemente in Diagramm 3), sondern fokusieren mehr die statischen Aspekte bzw. die durch die Ausführung eines Softwaresystems gewonnenen „dynamischen“ Informationen.

Der Quellcode liegt im „Source Repository“ vor. Parser/Analysierer, die verschiedenste Metriken bezüglich eines Softwaresystems gewinnen können, generieren Analysedaten, die der statische Visualisierer abrufen kann (Diagramm 2).

Die Erweiterung des Basisprozesses wird in Diagramm3 veranschaulicht, und wird in den Fällen vorgenommen, in denen Metriken bezüglich des Softwaresystems nötig sind, die erst durch die Ausführung des Systems gewonnen werden können.

Hierzu zählen beispielsweise „Profiling Informations“ (z.B. wie oft wird eine Zeile ausgeführt). Dafür ist es nötig, den Quellcode durch sogenannte „Probes“ zu instrumentieren, die vor jeder Ausführung des zu beobachtenden Quellcodeabschnittes dessen unmittelbare Ausführung in Logfiles festhalten. Die so gesammelten Informationen können anschliessend durch den statischen bzw. dynami-

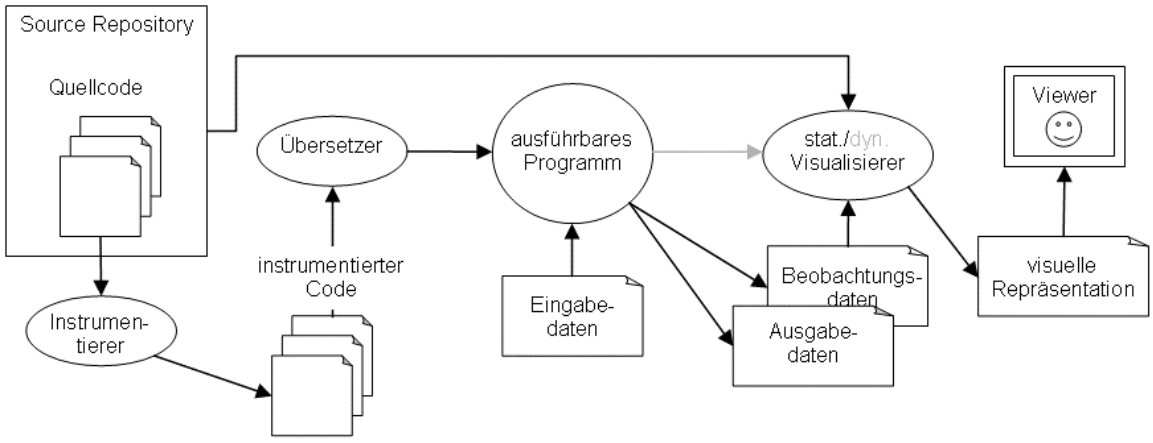


Diagramm 3, [110]
Erweiterte statische bzw. dynamische Softwarevisualisierung.

schen Visualisierer abgerufen werden.

Softwarevisualisierung wird daher in unterschiedlichsten Formen und Varianten, je nach konkreter Anforderung/Facette, realisiert und angewendet. Im folgenden Abschnitt wird die „Codezeilen-basierte Softwarevisualisierung“ (CZB SV) vorgestellt.

6.2 Codezeilen-basierte Softwarevisualisierung

Grundlage der CZB SV ist „die“ konkrete Quellcodezeile eines Softwaresystems.

Für verschiedene Zwecke kommt es darauf an, Metriken, die bezüglich einer konkreten Quellcodezeile („source code line level“) gewonnen werden, zu beurteilen bzw. für die Entscheidungsfindung, beispielsweise bezüglich der weiteren Planung eines Softwaresystems, heranzuziehen. Solche Metriken könnten sein [28]:

- welche Quellcodezeile(n) wurde(n) modifiziert (z.B. bezüglich eines „Modification Requests“)
- Zeitpunkt der Modifikation
- Person die Modifikation vorgenommen hat
- Art der Modifikation
- „Profiling Informations“, z.B. wie oft wird eine Quellcodezeile ausgeführt

Versionskontrollsysteme (z.B. CVS, SourceSafe, SubVersion, etc.) und Projektmanagementsysteme liefern diese Art von Metriken „frei Haus“, da sie jede Änderung, die bezüglich einer Datei gemacht wird, speichern.

Weiterhin ist für verschiedene Zwecke von besonderem Interesse die Verteilung der aktuell betrachteten Metriken innerhalb der Quellcodedatei bzw. über Quellcodedateigrenzen hinweg im gesamten Softwaresystem.

Aufgrund dieser beiden Anforderungen basiert die CZB SV auf 4 Kernideen [28]:

- **„reduced representation“**
  Aufgrund des grossen Volumen an Informationen bei komplexen Softwaresystemen, ist eine reduzierte Darstellung dieser Informationen nötig, um möglichst viel des Gesamtsystems sehen zu können („big picture“)
- **„coloring by statistics“**
  Unterscheidungsmöglichkeit von Metrikwerten ist durch die Vergabe von Farben am besten gegeben
- **„direct manipulation“**
  Manipulation und Selektion der Informationen auf dem Display zur Einschränkung der zu visualisierenden Informationsmenge bzw. zur Findung interessanter Muster
- **„capability to read actual code“**
  Notwendigkeit neben dem „big picture“ der „reduced representation“ zu jedem Zeitpunkt auch detailliertere Ansichten, die “finer grained details”, des gewählten Quellcodeabschnitts wählen zu können, bis hin zur Textdarstellung des Quellcodes

Die Basisumsetzung der Kernideen ist in Bild 1 zu sehen (SeeSoft, 1992). Grundelemente der Visualisierung sind (für den 2D Bereich):

- **Linie**
  repräsentiert eine Quellcodezeile
- **Box (Rahmen)**
  repräsentiert eine Quellcodedatei
- **Farbe**
  repräsentiert den Metrikwert, der für die jeweilige Quellcodezeile zutrifft
- **„Magnifying Boxes“/Textbrowser**
  erlauben detailliertere Ansichten, und die Betrachtung des konkreten Quellcodes

Es sei betont, dass es bei der CZB SV von besonderer Wichtigkeit ist, möglichst viel vom Gesamtsystem auf dem Display abzubilden, um so über das „big picture“ leichter Einsichten über das Softwaresystem gewinnen zu können.

Bild 1, [5]
Basisumsetzung der 4 Kernideen.

6.2.1 CZB SV in 2D

Quellcode liegt in 2D vor. Um ein „natürliches“ Mapping vom Quellcode zur Visualisierung und umgekehrt zu schaffen, bediente man sich einer „zoomed away“ Darstellung um die Vorgabe einer „reduced representation“ zu erreichen.

Durch eine extreme „zoomed away“ Darstellung, wird eine Quellcodezeile nicht mehr mit Hilfe von Zeichen, sondern nur noch als Linie mit einer Dicke von einem Pixel und einer Länge von mehreren Pixeln, entsprechend dem Verhältnis zur Länge der tatsächlichen Quellcodezeile, dargestellt.

Diese Darstellung erst ermöglicht das so wichtige „big picture“, die Möglichkeit möglichst viel des Gesamtsystems auf einem Blick betrachten zu können, und die Verteilung der gegenwärtig gewählten Metrik analysieren zu können.

„Line Representation“:
Jede Quellcodezeile wird durch eine Linie repräsentiert, die im Verhältnis zur jeweiligen Länge der Quellcodezeile steht (Bild 3, jeweils li. in den Hälften). Ebenso lässt sich die Einrückung der Linien wahlweise ein- oder abschalten, wobei im abgeschalteten Modus die gesamte Breite der umrahmenden Box für eine Linie ausgenutzt wird. Durch die „Line Representation“ lässt sich ein „natürliches“ Mapping von Quellcode zur Visualisierung und umgekehrt realisieren.

„Pixel Representation“:
Aufgrund des limitierten Platzes auf dem Display, lässt sich durch die „Pixel Representation“ mehr Quellcode und damit mehr Informationen abbilden (Bild 3, jeweils re. In den Hälften). Jeder Quellcodezeilenlinie wird eine feste Anzahl von Pixeln zugewiesen (z.B. 5). Die nicht mehr im Verhältnis zum Quellcode stehenden stellvertretenden Quellcodezeilenlinien werden nebeneinander, die gesamte Breite der umrahmenden Box ausnutzend, angeordnet.

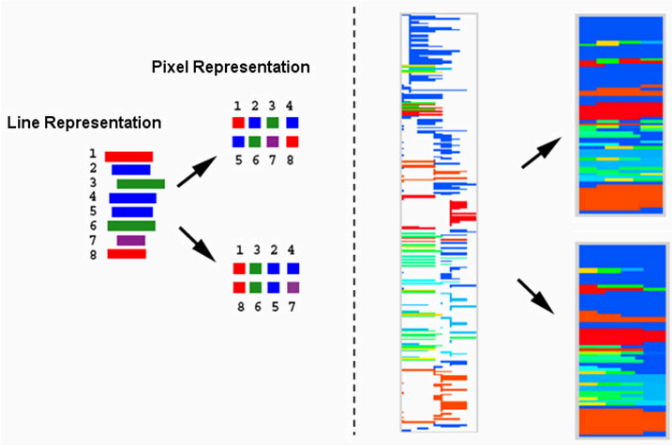


Bild 3, [5]
„Line Representation“ vs. „Pixel Representation“.

Dadurch lassen sich mehrere Quellcodezeilen in einer Zeile darstellen. In der „Pixel Representation“ kommt es dann nicht mehr auf die unmittelbare Identifizierung einer konkreten Quellcodezeile an, sondern vielmehr auf die Erkennung der jeweiligen farblich hervorgehobenen Metrik und deren Verteilung innerhalb der Quellcodedatei bzw. über Quellcodedateigrenzen hinweg. Innerhalb einer Zeile der „Pixel Representation“ liesse sich bei-

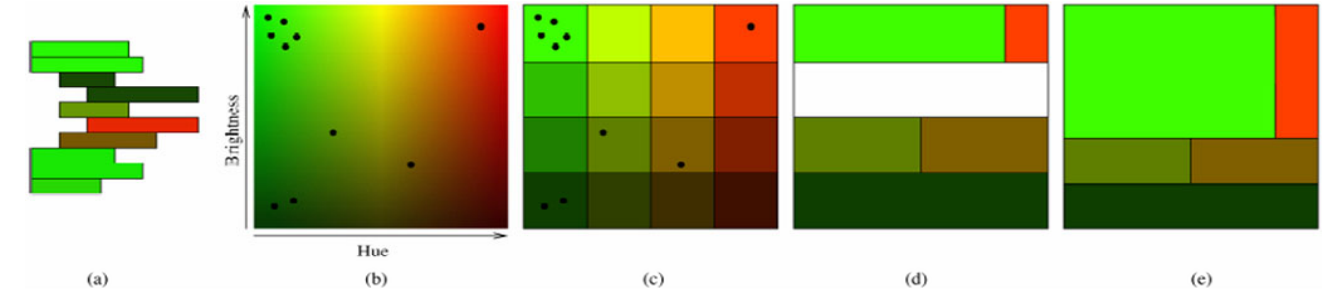


Bild 5, [44]
Treemap-Algorithmus.

spielsweise auch nach einem konkreten Metrikwert sortieren.

Durch die einfachste bzw. historisch gesehen erste Form der Implementierung der CZB SV mit SeeSoft, ist es leider so, dass nur eine Metrik zu einem Zeitpunkt betrachtet werden kann.

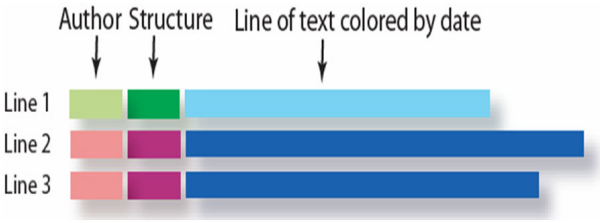


Bild 4, [22,23]
Augur, 3 Metriken gleichzeitig.

Neuere Systeme, beispielsweise Augur (Bild 4) [22,23], umgehen diese Einschränkung, indem sie vor der eingefärbten Quellcodezeilenlinie, zwei in der Länge verkürzte Linien reservieren, die für 2 weitere Metriken verwendet werden können. Somit können in der Summe 3 Metriken bezüglich einer Quellcodezeile gleichzeitig evaluiert werden.

In Kauf genommen muss dafür allerdings eine Verkleinerung der insgesamt darstellbaren Informationsmenge.

Aufgrund der Komplexität von Softwaresystemen, trotz Anwendung der „reduced representation“, reicht der verfügbare Platz auf dem Display nicht aus, um das „big picture“ des Gesamtsystems zu visualisieren.

Eine Erweiterung ist in Bild 5 zu sehen, basierend auf dem Tarantula/GAMMATELLA Treemap-Algorithmus [44].

In (a) liegt die „reduced representation“ vor, aus der die Treemap-Darstellung entwickelt werden soll. Dafür werden die Farbwerte jeder Quellcodezeile in den 2D-Farbraum, gegeben durch jeweils Achsen für den Farbwert und die Helligkeit, geplottet (b). Jeder Plottpunkt repräsentiert eine Quellcodezeile.

Anschliessend findet eine Aufteilung des Farbraums in gleichgrosse quadratische Bereiche statt (c); ein möglichst feines Gitter ist für eine genauere Visualisierung zu empfehlen. Der Repräsentativfarbwert für einen jeweiligen Bereich wird durch den Medianfarbwert eines jeweiligen Bereichs ermittelt.

In (d) und (e) findet die Vergabe der Breite und Höhe eines jeweiligen Bereichs statt, abhängig von der Anzahl der im jeweiligen Bereich geplotteten Quellcodezeilen, relativ zur Gesamtanzahl der geplotteten Quellcodezeilen.

Die CZB SV stellt sämtliche Werte der betrachteten Metrik in einer Maus-sensitiven Farbskala/-legende zur Verfügung, sodass der User einen bestimmten Wert einfach selektieren kann.

6.2 CZB SV in 3D

Einige wenige CZB SV-Tools (siehe 5.4.1) sind in 3D implementiert. Man versucht den Vorteil, den 3D Visualisierung bietet, für die CZB SV zu nutzen. Insbesondere sind dabei zu nennen:

- mehr Platz für die Anordnung der Metriken steht zur Verfügung
- der Mensch ist „aggieren“ in 3D gewohnt, es ist ihm aus der realen Welt bekannt

Eine mögliche resultierende Darstellung ist in Bild 6 zu sehen. Die Position eines Polyzylinders (repräsentiert eine Quellcodezeile) innerhalb des Containers (Plattform, repräsentiert eine Datei) ist abhängig von der Position der entsprechenden Quellcodezeile innerhalb der zugehörigen Quellcodedatei.

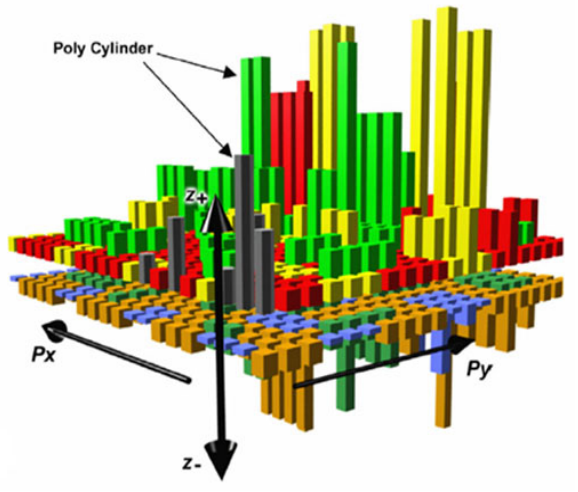


Bild 6, [70,71]
Codezeilen-basierte Softwarevisualisierung in 3D, Source Viewer 3D (sv3D).

Die zur Visualisierung verfügbaren Visualisierungselemente sind im Falle von sv3D

- Container o
- Polyzylinder p
- Position p_x
- Position p_y
- Höhe z_+
- Tiefe z_-
- Farbe c_+
- Farbe c_-
- Form des Polyzylinders d

sodass ein Visualisierungselement v_i ein 9-Tupel

$$v_i = \{o,p,p_x,p_y,z_+,z_-,c_+,c_-,d\}$$

ist, wobei die ersten 4 Elemente für den Polyzylinder und seine Positionierung innerhalb des Containers reserviert sind, und die folgenden 4 Elemente für Metriken und deren Werte vergeben werden können.

Visualisierung in 3D braucht Manipulations- und Interaktionstechniken. Von besonderem Interesse sind hierbei die Techniken, die von den „üblichen" abweichen:

- **„Rotation"**, die bezüglich der Achsen x, y, z durchgeführt wird
- **„Panning"**, das Platzieren des selektierten Objekts innerhalb einer 2D Ebene
- **„Zooming"**, das Vergrössern/Verkleinern der Szene

Manipulations- und Interaktionstechniken auf Objektebene bringen hierbei den entscheidenen Vorteil und den Unterschied, da sie ein typisches Problem in der 3D Visualisierung handhaben können: „Occlusion" (Verdeckung von Objekten). Zu nennen sind hierbei:

- **„Elevation",** das Positionieren einzelner Polyzylinder(gruppen), zu einer Metrik gehörend, entlang der z-Achsen zur besseren Einsicht (Bild 7).
- **„Simultaneous Alternative Mapping"** , das Erstellen einer/mehrfacher Kopien der Visualisierung, und unter Umständen Neubelegung der zu visualisierenden Metriken auf die verfügbaren Attribute innerhalb einer Kopie. Kernidee ist die gleichzeitige Betrachtung verschiedener Belegungen von Metriken des ein und selben Datensatzes. In Bild 8 visualisiert eine der Kopien nur noch die Metrik, die im Original auf z_+ des Polyzylinders gemappt war. Die zweite Kopie hingegen nun in z_+ nur noch das, was im Original auf z_- gemappt war.
- **„Transparenz",** das Durchsichtigmachen von Polyzylindern, um so den Fokus auf die nichttransparenten Polyzylinder mit bestimmten Metriken zu lenken; insbesondere falls diese Polyzylinder-menge in der Normalsicht verdeckt ist (Bild 9).
- **„Scaling/Stretching, Rotation",** das Skalieren und Verzerren der Visualisierung über verschiedene Angriffspunkte der „Handle Box" (Bild 10 li., gelber Rahmen) und die Rotation entlang der Achsen x,y,z über die Rollbügel des „Track Balls" (Bild 10 re., grau)

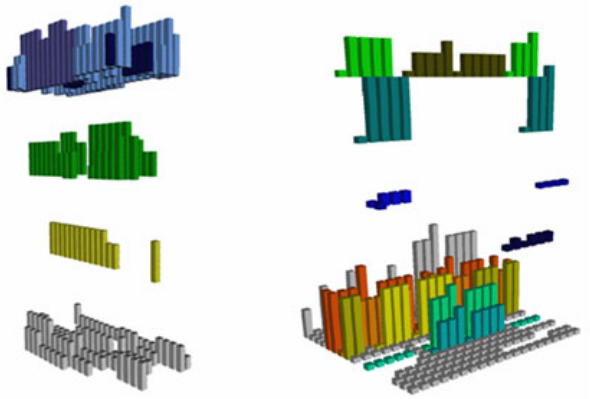


Bild 7, [70,71]: Elevation.

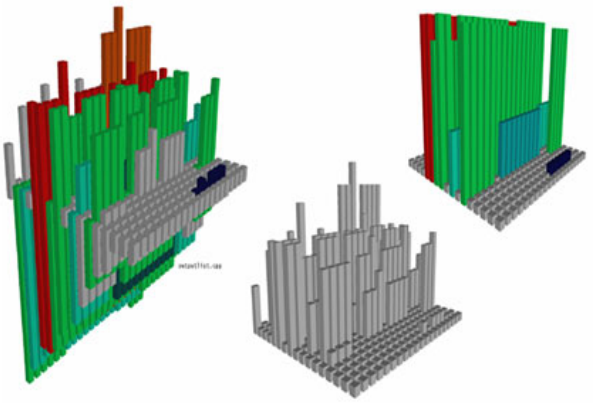


Bild 8, [70]: Simultaneous Alternative Mapping.

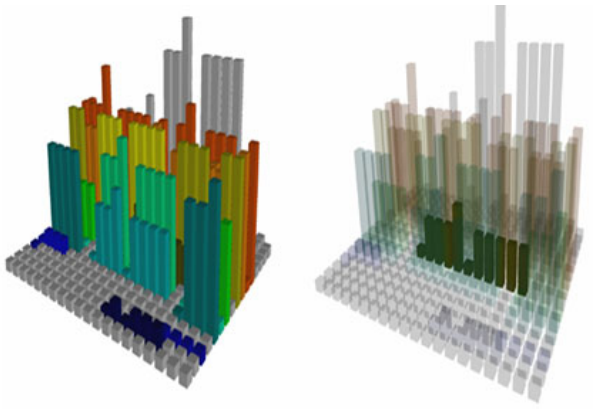


Bild 9, [70]: Transparenz.

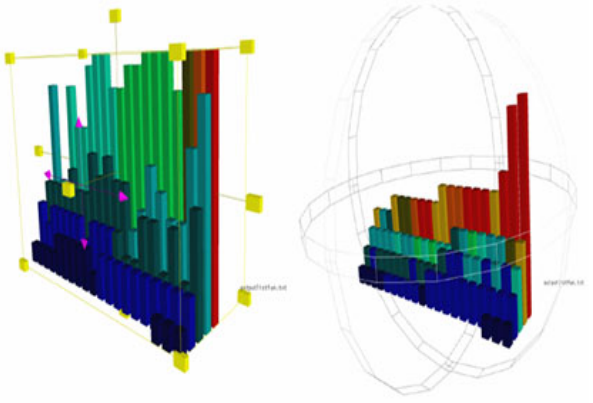


Bild 10, [70]: Scaling/Stretching, Rotation.

6.3 Einordnung und Bewertung

6.3.1 Abbildungsmächtigkeit in bezug auf Systemmodelle von UML

6.3.1.1. Abbildung der Elemente der statischen Systemarchitektur

Durch vorliegende Tools nicht eindeutig spezifiziert, aber durchaus realisierbar, da es zum einen abhängig ist von den eingesetzten Parsern/Analysierern und zum anderen von der programmiertechnischen Implementierung des Tools.

6.3.1.2. Abbildung der Elemente der dynamischen Systemarchitektur

Durch die erweiterte statische/dynamische Visualisierung (Diagramm 3) könnte eine Abbildung der Elemente der dynamischen Systemarchitektur realisiert werden.

6.3.1.3. Abbildung der Elemente der physischen Systemarchitektur

Durch vorliegende Tools nicht eindeutig spezifiziert, aber durchaus realisierbar, da es zum einen abhänging ist von den eingesetzten Parsern/Analysierern und zum anderen von der programmiertechnischen Implementierung des Tools.

6.3.2 Klassifikation des Software-Visualisierungsansatzes

6.3.2.1. Visualisierungsprinzip

In 2D:

Die Umsetzung der 4 Kernideen [28] der CZB SV besteht in der Verwendung von Linien, Farben und Boxen (Rahmen). (s. 2.1)

In 3D:

Die Umsetzung CZB SV besteht im Falle von sv3D [69,70,71] in der Verwendung von Polyzylindern, Farben und Containern (Plattformen). (s. 2.2)

6.3.2.2. Visualisierungsprozess

Ein konkreter Anwendungsfall (für CZB SV in 2D) des in Diagramm 3 beschriebenen Visualisierungsprozesses ist in Diagramm 4 beschrieben.

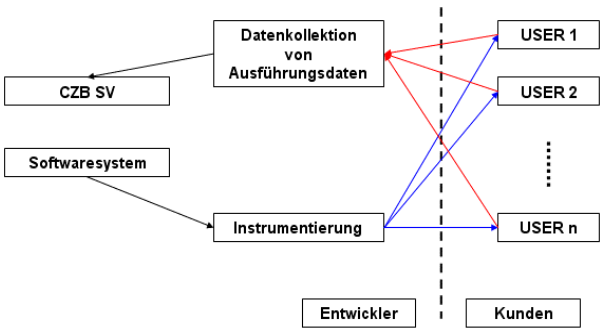


Diagramm 4, [44]
GAMMATELLA, Prozess der Datengewinnung und Visualisierung.

GAMMATELLA basiert auf Tarantula, einem CZB SV-Tool zur Visualisierung von Quellcodezeilen, die aufgrund von Ausführungsdaten einer Testsuite, als fehlerhaft einzustufen sind. In GAMMATELLA ist die Erweiterung implemen-

tiert, die zu untersuchende Software über „Remote Monitoring“ beim Kunden vor Ort auszuführen und die dort gewonnenen Ausführungsdaten auf Entwicklerseite gesammelt zwecks Fehlerbeseitigung zu visualisieren.

Ein konkreter Anwendungsfall (für CZB SV in 3D) des in Diagramm 2 beschriebenen Visualisierungsprozesses ist in Diagramm 5 beschrieben.

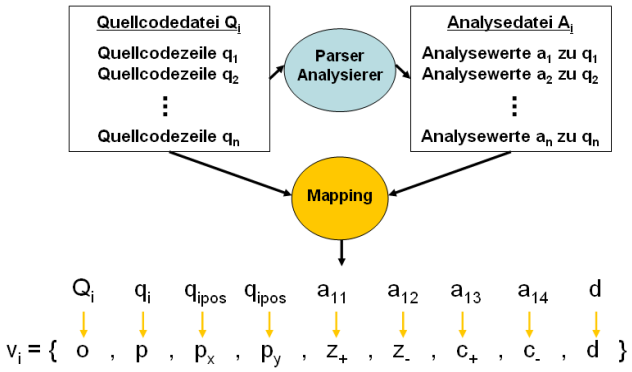


Diagramm 5, [70,71]
sv3D, Prozess der Datengewinnung und Visualisierung.

Zu jeder Quellcodezeile q_k einer Quellcodedatei Q_i liefern Parser/Analysierer Analysewerte $a_k = \{a_{k1}, a_{k2}, \ldots, a_{km}\}$ in einer Analysedatei A_i im XML-Format. Das darauf folgende Mapping von sv3D bildet die zu visualisierenden Werte auf die verfügbaren Visualisierungselemente (s. 2.2). sv3D setzt beim Input-Format auf XML, um so formatunabhäging von den verschiedenen auf dem Markt verfübaren Parsern/Analysierern zu sein. Eine Transformation der Outputformate verschiedener Hersteller nach sv3D XML-Inputformat ist möglich.

6.3.2.3. Intention des Ansatzes

Ursprünglich ist die CZB SV eingeführt worden, um die, durch die Versionskontrollsysteme gegebenen Historiendaten zu visualisieren. Diese Daten eignen sich zum einen, je nach gewählter Metrik, sowohl zur Erstanalyse des Systems als auch zur Betrachtung der inkrementellen Änderungen.

6.3.2.4. Unterstützte Programmiersprachen

Die unterstützten Programmiersprachen variieren von C/C++, über Cobol, zu Java. Sicherlich lässt sich eine genaue Aussage über die unterstützten Programmiersprachen machen, in dem der Markt an Parsern/Analysieren studiert wird, da diese die Analysedaten bezüglich eines Systems liefern. Ebenso verhält es sich mit der Verfügbarkeit von Instrumentierern für verschiedenste Programmiersprachen.

6.3.2.5. Abhängigkeit von Programmierkonventionen

Spezielle Elemente, die dem Quellcode hinzugefügt werden müssen um die Visualisierung im Falle von Diagramm 3 zu ermöglichen, werden über die Instrumentierer (z.B. InsECT) bereitgestellt. Es handelt sich dabei um sogenannte „Probes“, Anweisungen, die unmittelbar vor der Ausführung eines beobachteten Quellcode-abschnitts, dies in Logfiles festhalten.

Zu bemerken ist hierbei, dass die Instrumentierung eines Softwaresystems eine Verlangsamung der Ausführungsgeschwindigkeit des Softwaresystems zur Folge hat.

Ein konkretes Rechenbeispiel ist in [44] gegeben: die Instrumentierung des Softwaresystems JABA, ca. 550 Quellcodedateien mit ca. 60.000 Quellcodezeilen, mittels InsECT, hatte eine Reduzierung der Ausführungsgeschwindigkeit von JABA um ca. 30% zur Folge.

Die Instrumentierungsdauer eines Softwaresystems ist zum einen abhängig von der Grösse des zu instrumentierenden Softwaresystems, zum anderen von der verfügbaren Rechenleistung (CPU, RAM, OS, etc.) aber auch von dem eingesetzten Instrumentierer, und kann einige Minute bis mehrer Stunden dauern.

6.3.2.6. Skalierbarkeit in bezug auf die Systemgrösse

CZB SV ist bezüglich der zu analysierenden Systemgrösse nicht eingeschränkt. Indirekt ist sehr wohl eine Einschränkung gegeben, denn ein Display ist nur begrenzt gross.

Man rechnet bei der CZB SV in 2D mit ca. 50.000 darstellbaren Quellcodezeilen auf einem Display, in der „Line Representation“, bei einer Auflösung von 1280x1024 Pixeln; in der „Pixel Representation“ ca. 1.000.000 Quellcodezeilen.

Zum Einsatz gekommen ist diese Art der Softwarevisualisierung auch in sehr grossen Softwareprojekten (5ESS, Bell-Labs) mit mehreren Millionen Zeilen Quellcode und einem Entwicklungszeitraum von ca. zwei Jahrzehnten mit mehreren tausend Entwicklern.

Nach wie vor besteht ein Problem bei der Visualisierung grosser Softwaresysteme bei den 3D Lösungen hinsichtlich der Performance beim Rendering und der Userinteraktion. Derzeit lassen sich mit sv3D 40.000 – 50.000 Quellcodezeilen gut darstellen; in der 2D Übersicht sogar ca. 100.000 Quellcodezeilen.

6.3.2.7. Automatisierungsgrad

Der manuelle Anteil ist eher als gering einzuschätzen, was die Visualisierung ansich betrifft, also die Anwendung des Tools auf die durch Par-

Tool	Jahr	Forscher/Institution	Funktionalität
SeeSoft	1992	Eick, Steffen, Sumner; AT&T Bell Labs	2D line oriented source code representation, coloring of statistics, 1 attribute at a time
Aspect Browser	1999	Griswold;University of California, San Diego Kato, Yuan; University of Tokyo	2D line oriented source code representation, coloring of aspects
Tarantula	2001	Eagan, Harrold, Jones, Stasko; Georgia Tech University	2D line oriented source code representation, coloring of execution data/statistics, 1 attribute at a time
Aspect Mining Tool	2001	Hannemann, Kiczales; University of British Columbia	2D line oriented source code representation, coloring of statistics
BLOOM: Bee/Hive	2001	Reiss; Brown University	2D/3D line oriented source code representation, coloring of run time spent in a routine
GAMMATELLA	2003	Harrold, Jones, Orso; Georgia Tech University	2D line oriented source code representation, coloring of execution data/statistics, 1 attribute at a time
sv3D	2003	Maletic, Feng; Kent State University Marcus; Wayne State University	3D line oriented source code represenentation, coloring of statistics, up to 4 attributes at a time
Augur	2004	Froehlich, Dourish; University of California, Irvine	2D line oriented source code representation, coloring of statistics, 3 attributes at a time
Cleanscape Testwise	1999	Cleanscape Product release	2D line oriented source code representation, coloring of statistics

Tabelle 1
Übersicht einiger Implementierungen der Codezeilen-basierten Visualisierungstechnik

ser/Analysierer verfügbar gemachten Metriken eines Softwaresystems.

Im Falle der erweiterten statischen Visualisierung (Diagramm 3), müssten natürlich erst Informationen durch Ausführung des Softwaresystems generiert werden, was unter Umständen einige Zeit in Anspruch nehmen kann (einige Minuten – mehrere Monate), abhängig sicherlich von der „Genauigkeit“ die man erzielt haben möchte. Die Instrumentierung des Quellcodes zählt sicherlich als eine manuelle Vorarbeit, auch wenn die Instrumentierung dann automatisiert durchgeführt wird.

6.3.2.8. Referenzen

- 5ESS, Bell Labs [28]
- Softwaresysteme die hinsichtlich des Y2K-Problems korrigiert werden mussten [30]
- Analyse verschiedener Implementierungen von WebServices [73]
- „Profiling Informations“ verschiedener Softwaresysteme [44]

6.3.3 Werkzeugeigenschaften

6.3.3.1. Einbettung in eine spezifische Entwicklungsumgebung

CZB SV Tools werden derzeit in keine der in der Praxis befindlichen Entwicklungsumgebungen eingebettet. Weiterhin handelt es sich zunächst nur um reine Visualisierer, es werden also keine Manipulationen des Quellcodes zugelassen. Sie liegen derzeit als Standalone-Tools vor.

Zukünftige Erweiterungen sollen dahingehend realisiert werden, dass der zugrundeliegende Quellcode „on-the-fly“ geändert werden kann. Dann liese sich sicherlich auch eine sinnvolle Einbettung realisieren.

6.3.3.2. Produktform

Softwarevisualisierung ist Gegenstand der Forschung, somit befinden sich die meisten Systeme nicht in einem Produktstatus, sie sind in der Regel noch nicht einmal zum Download verfügbar. Ausnahmen bilden „lauffähige“ Versionen von Augur, bzw. das als Produkt verfügbare „Cleanscape Testwise“. Sämtliche Rechte liegen bei den jeweiligen Institutionen.

6.3.3.3. Unterstützte Plattform

Unix/Linux und Windows; wenn in Java implementiert sogar plattformunabhängig.

6.3.4 Forschung und Entwicklung

6.3.4.1. Beteiligte Institutionen und Forscher

s. Tabelle 1

6.4 Zusammenfassung

CZB SV ist ein mächtiges Werkzeug zur Unterstützung des Verstehensprozesses eines Softwaresystems. Die Entwickler des ersten CZB SV-

Systems, SeeSoft, sehen verschiedenste Einsatzbereiche [28], die auch nach 1,5 Jahrzehnten in dieser allgemeinen Form ihre Gültigkeit haben:

- Verstehen und Erforschen des Codes
- Training neuer Entwickler
- Projekt-Management
- Qualitätssicherung und Systemtest
- Softwareanalyse
- Optimierung des Codes

CZB SV ist nach wie vor Gegenstand aktueller Forschung, obwohl es seit 1992 kontinuierlich neue Variationen für verschiedenste Einsatzbereiche (s. Tabelle 1, 3.2.8) gegeben hat und weiterhin geben wird.

Die CZB SV besticht durch ihre einfache und durch den User leicht aufnehmbare Visualisierungstechnik, deren Einsatz in zukünftigen Entwicklungsumgebungen erwartet wird.

CZB SV ist derzeit nicht dafür ausgerichtet hierarchische Strukturen zu visualisieren, weder im 2D noch im 3D Bereich. Primärer Fokus der Entwickler des ersten CZB SV-Tools SeeSoft, war es C basierte Softwaresysteme zu visualisieren, so dass es nicht verwunderlich ist, dass die CZB SV nicht für hierarchische Strukturen optimiert ist. Programmiertechnisch liesse sich sicherlich das ein oder andere realisieren, ist aber ursprünglich nicht angedacht gewesen.

Eine Weiterentwickler für den 3D Bereich, im Falle von sv3D, ist dahingehend geplant, einzelne Container zu verlinken, sodass diese Einschränkung aufgehoben werden kann. Gerade die CZB SV in 3D würde sich für die Visualisierung hierarchischer Strukturen eignen, aufgrund des „unbegrenzten“ Platzes und der Möglichkeit Elemente flexibler, in verschiedenen Ebenen anzuordnen.

Ein schon bekanntes Problem würde dann allerdings wieder in den Mittelpunkt des Geschehens rücken: das sinnvolle Layouten der Elemente im 3D Raum, was innovative Manipulations- und Interaktionstechniken zur Folge haben müsste.

Kapitel 7

Landschafts- und Stadtmetaphern zur Softwarevisualisierung

Benjamin Hagedorn

Kapitel 7: Landschafts- und Stadtmetaphern zur Softwarevisualisierung

Benjamin Hagedorn

Zusammenfassung. Softwaresysteme sind sehr komplexe Systeme und wegen ihrer intangibelen Natur oft nur schwer begreifbar. Im Bereich der Informationsvisualisierung existieren viele Ansätze um die Entwicklung und Wartung von Software durch eine angemessene Darstellung zu unterstützen. Eine besondere Bedeutung für diese Darstellung können Metaphern haben, da sie die vielfältigen und hochkomplexen Informationen über das Softwaresystem in einer für den Benutzer intuitiven Form darzustellen versuchen. Dieser Artikel gibt einen Überblick über die Verwendung der Landschaftsmetapher und der Stadtmetapher zur Softwarevisualisierung. Wir beschäftigen uns mit den Elementen der Metaphern und der Abbildung der Informationen auf diese. Außerdem werden weitere Aspekte wie die angemessene Orientierungsunterstützung und Navigation betrachtet. An verschiedenen Beispielen wird der Stand der Forschung auf diesem Bereich dargestellt und diskutiert.

7.1 Einführung

Softwarevisualisierung ist auf Softwaresysteme bezogene Informationsvisualisierung. Nach [56] ist Softwarevisualisierung „[…] a discipline that makes use of various forms of imagery to provide insight and understanding and to reduce complexity of the existing software system under consideration."[5]

Softwarevisualisierungssysteme haben die Aufgabe, den Prozess der Verständnisgewinnung über ein Softwaresystem zu unterstützen. [57] bezeichnen Softwarevisualisierungssysteme deshalb als Werkzeuge zur „Intelligence Amplification", die den Menschen in seinen besonderen Fähigkeiten unterstützen. Diese Fähigkeiten sind die gute Erkennung von Mustern, die schnelle Beurteilung von Sachverhalten und das ausgeprägte Kontextverständnis.

Software ist intangibel und jeder Programmierer hat ein eigenes mentales Modell über das System. Hier kann Softwarevisualisierung eine einheitliche Basis für die effizientere Kommunikation über die Software bieten.

[72] zählen folgende fünf Dimensionen auf, an denen sich jedes Softwarevisualisierungssystem ausrichtet:

Ziel: Das Ziel definiert welche Daten durch das Visualisierungssystem dargestellt werden sollen – ob das ganze Systeme oder nur Teile betrachtet werden, ob nur die Beziehungen von Interesse sind, oder ob zum Beispiel nur Kosten dargestellt werden sollen.

Nutzer: Je nach Nutzer sollte die Visualisierung unterschiedliche Aspekte hervorheben. Nutzer können zum Beispiel Programmierer, Projektleiter, Manager verschiedener Ebenen oder der Kunde sein.

Aufgabe: Die Aufgabe definiert den Zweck der Visualisierung. Mögliche Aufgaben sind Reverse-Engineering, Entwicklung, Wartung oder Controlling.

Repräsentation: Repräsentationen sind die grafischen Elemente mit denen die Softwareinformationen dargestellt werden sollen. Mögliche Repräsentationen sind Graphen, Würfel oder Städte.

Medium: Das Medium beschreibt, womit die Darstellung erfolgen soll – ob dies in 2-D oder 3-D geschehen soll, ob als Träger Papier, ein Bildschirm oder ein CAVE[6] verwendet wird.

Bei der Betrachtung von Softwarevisualisierungssystemen müssen verschiedene Aspekte betrachtet werden. Einige wichtige sind (nach [56]):

Metapher, Mapping, Repräsentation, Visualisierung: Metaphern sind Konzepte zur Abbildung komplexer Sachverhalte auf einfacher strukturierte, besser bekannte und besser nachvollziehbare Gegenstandsbereiche – hier also die Übertragung des Systems der Softwareinformationen ins Bildhafte. Das Mapping ist die konkrete Abbildung der Softwareinformationen auf einzelne Repräsentationen. Diese Repräsentationen sind grafische Elemente oder deren Merkmale. Eine Visualisierung ist nun eine konkrete Menge solcher Repräsentationen und ihrer Anordnung.

Interaktion, Navigation, Orientierung: Die Interaktion mit der Visualisierung bestimmt we-

[5] [Knight2000], S. 40.

[6] Cave Automatic Virtual Environment

sentlich den Benutzungskomfort des Visualisierungssystems. Interaktion beinhaltet sowohl eine möglichst einfache und intuitive Navigation als auch Abfrage- und Auswahlmöglichkeiten sowie Möglichkeiten zur Manipulation der dargestellten Inhalte. Ohne gute Navigationsmittel ist eine weiterreichende Interaktion mit Darstellungselementen nicht möglich. Das Visualisierungssystem muss dem Nutzer zudem geeignete Orientierungshilfen zur Verfügung stellen, damit dieser sich in der Visualisierung nicht verirrt.

Abstraktion: Abstraktion ist hier die Darstellung in einer anderen Detaillierungsstufe: Statt der Klassenebene wird zum Beispiel die Paketebene betrachtet.

Animation: Animation kann sowohl zur Darstellung von Software-Evolution als auch zur Abbildung von Dynamik verwendet werden. Dabei treten zum einen große Datenmengen und zum anderen in hoher Rate immer neue Informationen auf, was zu erhöhten Anforderungen an die Visualisierung führt. Zum Beispiel müssen optische Brüche während der Darstellungszeit verhindert werden, die dem Benutzer die Orientierung erschweren.

Skalierbarkeit: Komplexe Software ist zumeist auch sehr umfangreich. Deshalb müssen Softwarevisualisierungssysteme mit großen Softwareprodukten umgehen können. Dazu zählen die effiziente Handhabung der anfallenden Softwareinformationen und deren ressourcenschonende Umsetzung in die durch die Grafikhardware darzustellende Visualisierung.

Automatisierung: Der Grad der Automatisierung bei der Erstellung der Repräsentationen und der Visualisierung ist besonders wichtig, wenn die Visualisierung schnell erzeugt werden soll oder wenn große Mengen an einzubindenden Softwareinformationen vorhanden sind. Auf der anderen Seite kann die Beteiligung des Benutzers bei der Erstellung der Visualisierung zu einer besseren Identifizierung mit der Visualisierung und zu einem verbesserten Verständnis führen. Die Automatisierung beinhaltet zum Beispiel das Layout der Repräsentationen, die Wahrnehmbarkeit der Visualisierung oder auch die Robustheit gegen kleine Änderungen der zu Grunde liegenden Softwareinformationen.

7.2 2-D-, 3-D- oder 2,5-D

Eine 2-D-Darstellung erlaubt nur 2-dimensionale Elemente, die auch nur entlang zweier Achsen angeordnet werden können.

Im Gegensatz zu 2-D-Darstellungen haben 3-D-Darstellungen eine Reihe von Vorteilen. Sie bieten eine zusätzliche Raumdimension und damit geometrisch komplexere und vielfältigere Objekte. Die dritte Dimension erlaubt eine kompaktere Verteilung der Informationsträger, wodurch eine höhere Informationsdichte möglich ist. Durch die dritte Dimension ist es möglich, Verbindungen zwischen Elementen ohne Überschneidungen darzustellen. Insgesamt erlauben 3-D-Darstellungen somit eine höhere Übersichtlichkeit. Außerdem kann das subjektive Empfinden des Benutzers verbessert werden: Der Benutzer ist „mitten drin“ und wird leicht ein Teil dieser Welt, was zu einer intensiveren und verbesserten Wahrnehmung durch den Benutzer führen kann.

Die 3-dimensionale Darstellung wirft auch Probleme auf. Die Anordnung der Informationsträger im Dreidimensionalen erfordert komplexere Layout-Algorithmen. Die zusätzliche Dimension erfordert eine komplexere Navigation und eine aufwendigere Benutzerschnittstelle. Bei der Navigation im Dreidimensionalen ist zudem eine Desorientierung des Benutzers wahrscheinlicher. Hinzu kommt die Gefahr der Informationsüberflutung des Benutzers durch zu dichte, zu komplexe und zu vielfältige Informationen. Außerdem ist die 3-dimensionale Darstellung im Allgemeinen aufwendiger und teurer als eine reine 2-D-Visualisierung.

Im Besonderen das Problem der Orientierung im 3-dimensionalen Raum führte zum Ansatz der 2,5-dimensionalen Visualisierungen. Eine solche Darstellung nutzt ebenfalls 3-dimensionale Objekte zur Darstellung. Diese werden jedoch nicht frei im 3-D-Raum platziert, sondern werden immer auf einer Bezugsebene angeordnet. Dadurch werden dem Benutzer die Orientierung und die Navigation erleichtert.

Visualisierungen mit Hilfe von Landschafts- oder Stadtmetapher sind typische Vertreter solcher 2,5-D-Visualisierungen.

7.3 Metaphern

Metaphern entwickelten sich im Laufe der Geschichte vom rein sprachlichen Phänomen zu komplexen Systemen. Etwa ab 1980 bildete sich die Kognitive Metapherntheorie heraus. Sie beschreibt eine Metapher als Konzeptionalisierung eines abstrakten und komplexen Gegenstandsbereiches durch einen konkreten, einfacher strukturierten und erfahrungsnäheren Gegenstandsbereich. Laut [90] besitzen Metaphern drei wesentliche kognitive Funktionen:

Erklärungs- und Verständnisfunktion: Durch Konkretisierung von abstrakten Sachverhalten erlauben Metaphern das Verständnis von komplexen Gegenstandsbereichen.

Kreatives Potential: Metaphern können auch Anreiz sein zum Umdenken und zum Perspektivwechsel.

Fokussierungseffekt: Metaphern können bestimmte Aspekte eines Gegenstandsbereiches hervorheben und andere verbergen.

[57] schreiben, dass die Nützlichkeit der Verwendung einer bestimmten Metapher nur schwer nachweisbar ist, da die Aufgaben und die Benutzer sich sehr voneinander unterscheiden. Bei [26] wurde jedoch durch Experimente belegt, dass durch die Verwendung einer Metapher zur Informationsvisualisierung sowohl die Qualität der Antworten als auch die Leistungsfähigkeit der Probanden gesteigert werden konnte. (Nach [57].)

7.3.1 Metaphern zur Visualisierung von Software

Im Bereich der Softwarevisualisierung sind Metaphern also die Darstellung der Softwareinformationen durch eine Reihe von Elementen des entsprechenden Metaphern-Konzeptes.

In diesem Sinne verwendet jedes Softwarevisualisierungssystem eine oder mehrere Metaphern. Beispiele sind die Verwendung von Graphen, Sonnensystemen oder von Videospielen als Metaphernkonzepte.

7.3.2 Abstrakte Metaphern und Realwelt-Metaphern

Im Bereich der 3-D- oder 2,5-D-Softwarevisualisierung sind sowohl abstrakte als auch wirklichkeitsgetreue Metaphern denkbar.

Eine abstrakte Metapher bietet eine unbegrenzte Fülle von Visualisierungsobjekten und die Möglichkeit die Metapher möglichst gut für die Visualisierung von Software anzupassen. Auf der anderen Seite erschwert sie dem Benutzer den Zugang zu den dargestellten Objekten und damit auch zu ihrer Bedeutung. Er muss neben der Zuordnung der Informationen auf die grafischen Repräsentationen auch diese Repräsentationen selbst und deren Beziehungen zueinander kennen lernen.

Realwelt-Darstellungen entstammen dem direkten Gegenstandsbereich des Nutzers. Sie bieten ein intuitives Verständnis und fördern die Bereitschaft zur Auseinandersetzung mit der Visualisierung, denn der Nutzer ist mit dem Metaphernkonzept vertraut. Realwelt-Metaphern sind jedoch nicht so leicht erweiterbar und besitzen aus ihrer Natur heraus eine Menge von Restriktionen, die dem Benutzer den Zugang zu den Informationen erschweren können. Ein wichtiger Punkt ist die Komplexität der dargestellten Realwelt. Bei vielen Autoren herrscht die Meinung vor, dass der Benutzer nicht mit zu vielen dargestellten Elementen überladen werden soll. Dies würde zur Verwirrung führen und den Wahrnehmungsprozess signifikant erschweren. Auf der anderen Seite sollte die Darstellung auch nicht zu sehr stilisiert sein, denn sonst bestünde eine geringere Vertrautheit des Benutzers mit der Darstellung, die Wahrnehmung würde gestört und die Effizienz der Arbeit mit dem Visualisierungssystem verringert. Deshalb haben [78] zur Erhöhung der Realität ihrer Stadtmetapher Bäume und Straßenlaternen eingefügt.

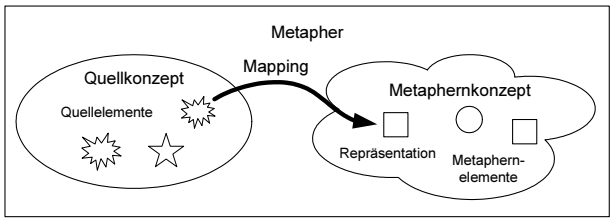


Abbildung 29: Metapher, Mapping, Repräsentation.

Es ist jedoch zu vermuten, dass eine Realwelt-Metapher die Bereitschaft fördert sich mit dem Visualisierungswerkzeug auseinanderzusetzen, denn durch die bekannten Objekte wird der Einarbeitungsaufwand vermindert und es können schnell strukturelle Informationen antizipiert werden.

7.3.3 Mapping, Repräsentation und Visualisierung

Abbildung 29 veranschaulicht den Zusammenhang zwischen der Metapher und der Erzeugung von Repräsentationen durch die Abbildung von Elementen des Quellkonzeptes auf Elemente des Metaphernkonzeptes. Im Rahmen des Metaphernkonzeptes erfolgt die Abbildung der Softwareinformationen in die grafische Form, die [109] in *Repräsentationen* und *Visualisierungen* unterscheiden. Repräsentationen sind grafische Komponenten, die verschiedene Softwareinformationen abbilden, während Visualisierungen bestimmte Mengen oder Konfigurationen von Repräsentationen sind, die gemeinsam eine Repräsentation einer höheren Abstraktion bilden.

Young und Munro haben wünschenswerte Eigenschaften von Repräsentationen und Visualisierungen identifiziert um die Softwarevisualisierung optimal nutzbar zu gestalten.

Erstrebenswerte Eigenschaften einer Repräsentation sind:

Individualität: Verschiedene Softwareinformationen erhalten verschiedene Repräsentationen. Andersherum weisen identische Repräsentationen auf identische Softwareinformationen hin.

Unterscheidbarkeit: Verschiedene Repräsentationen sollen so unterschiedlich wie möglich sein um die Wahrnehmung des Nutzers zu unterstützen.

Angemessene Komplexität: Die Repräsentation einer Softwarekomponente soll möglichst viele Informationen über die Komponente enthalten. Auf der anderen Seite soll die Repräsentation nicht übermäßig komplex sein um die Wahrnehmung des Nutzers und Leistung des Visualisierungssystems nicht zu behindern.

Skalierbarkeit der visuellen Komplexität und des Informationsgehaltes: Die Repräsentation einer Softwarekomponente soll sowohl detaillierte Betrachtungen als auch eine Übersicht über die Komponente ermöglichen.

Flexibilität für die Integration in Visualisierungen: Softwareinformationen sollen so durch grafische Elemente repräsentiert werden, dass die erzeugte Repräsentation einer Softwarekomponente gut in die Visualisierung integriert werden kann.

Automatisierbarkeit: Repräsentationen sollten so gestaltet sein, dass sie leicht, schnell und automatisch aus den Softwareinformationen abgeleitet werden können. Komplexe Formen und Ableitungsvorschriften können dies zum Beispiel behindern.

Erstrebenswerte Eigenschaften einer Visualisierung sind:

Angemessene Komplexität: Auch Visualisierungen sollen möglichst viele Informationen enthalten und zugleich gut wahrnehmbar sein.

Verschiedene Detaillierungsstufen: Dem Benutzer sollen verschiedene Detaillierungsstufen einer Visualisierung angeboten werden, damit dieser die Visualisierung entweder Top-Down oder Bottom-Up erkunden kann.

Änderungsinvarianz: Kleine Änderungen an den Softwareinformationen und damit auch an den Repräsentationen sollen keine großen Auswirkungen in der Visualisierung nach sich ziehen.

Gute Benutzerschnittstelle: Die Benutzerschnittstelle darf den Benutzer so wenig wie möglich behindern und muss so intuitiv wie möglich bedienbar sein, um die Aufmerksamkeit des Benutzers nicht unnötigerweise auf die Bedienung zu richten.

Einfache Navigation: Visualisierungen müssen benutzerfreundlich strukturiert sein, um eine Desorientierung bei der Navigation zu vermeiden.

Integration zusätzlicher Informationen: Neben den primären Softwareinformationen wie der Programmstruktur sollten auch andere Informationen wie zum Beispiel Quelltext oder Dokumentationen eingebunden werden können, um eine ganzheitliche Betrachtung des Softwaresystems zu ermöglichen.

Automatisierbarkeit: Auch die Visualisierungen sollten gut automatisch erzeugbar sein um die Softwarevisualisierungssysteme wirklich nutzbar zu machen.

[68] hat zwei Kriterien entwickelt, um die Abbildung von Informationen auf eine visuelle Metapher zu bewerten (nach [72]). Diese sind die *Ausdrücksstärke* der Metapher und ihre *Effektivität*. Die Ausdrucksstärke beschreibt die Fähigkeit der Metapher alle notwendigen Informationen zu repräsentieren. So müssen die Elemente der Metapher genügend visuelle Parameter besitzen, auf die die Softwareinformationen abgebildet werden können. Die Effektivität beschreibt die Qualität der Abbildung in die Metapher bezüglich der visuellen Wahrnehmbarkeit und Ästhetik der Darstellung. So müssen die verwendeten Metaphernelemente für die spezielle Abbildung auch wirklich geeignet sein. Zum Beispiel können quantitative Informationen gut durch das grafische Attribut Größe aber weniger gut durch die Form abgebildet werden. Die Effektivität einer Abbildung bezieht sich auch auf rechentechnische Optimierungen wie zum Beispiel die Verringerung der Anzahl der an die Grafikhardware zu übertragenden Objekte oder die Reduzierung ihrer grafischen Komplexität.

Im Besonderen die visuelle Effektivität der Abbildung erfordert die Kategorisierung der Quell- und Zieldaten, also der Softwareinformationen und der Informationen des Metaphernsystems. Sie können zum Beispiel nach ihrer Art oder ihrer Wichtigkeit unterschieden werden. Diese Einteilung ist nutzer- und aufgabenspezifisch und ermöglicht eine möglichst effektive Abbildung ins Metaphernkonzept.

Eine mögliche Einteilung der Softwareinformationen ist die in skalare Werte und Beziehungen. Skalare Werte könnten auf Form und Geometrie, Größe, Farbe und Transparenz, Texturen oder Animationen abgebildet werden. Beziehungen könnten auf die räumliche Nähe, die Anordnung, Ausrichtung, physikalische Verbindungen oder Container abgebildet werden.

Eine andere mögliche Einteilung der Softwareinformationen ist die in qualitative und quantitative Informationen, also in Aussagen über die Typen von Entitäten und Aussagen über Mengen bezüglich dieser Entitäten. Qualitative Informationen lassen sich gut durch Farbe und Form darstellen, während für quantitative Informationen die Größe der Repräsentation besser geeignet ist.

7.4 Landschaftsmetaphern und Stadtmetaphern

Es existieren verschiedene Ansätze zur Softwarevisualisierung mit Hilfe von Landschafts- und Stadtmetaphern. Diese gehen jedoch nicht über einen konzeptionellen Status hinaus. Es fehlen nachvollziehbare und überprüfbare Ergebnisse in Form von Prototypen oder Nutzerstudien. Auf dem Gebiet der 3-D-Softwarevisualisierung und speziell bei der Verwendung von Landschafts- oder Stadtmetaphern scheinen auch grundlegende Fragen noch nicht geklärt: Uns sind keine Untersuchungen darüber bekannt, welche grafischen Elemente einer Landschafts- oder Stadtmetapher Sinnvollerweise als Repräsentationen dienen können und wie deren Wirkung auf den Benutzer ist.

Im Folgenden beschreiben wir die Elemente der Landschafts- und der Stadtmetapher und betrachten die Anwendung der Metaphern bezüglich der für Softwarevisualisierung wichtigen Aspekte.

7.4.1 Repräsentationen der Landschaftsmetapher

Eine reale Landschaft wird durch ihre Oberflächenstruktur, ihre Morphologie und ihre Vegetation geprägt. Diese Merkmale könnten als Repräsentationen zur Softwarevisualisierung genutzt werden.

Beispiele für Metaphernelemente der Landschaftsmetapher sind folgende (nach [56]):

Morphologische Merkmale: Kliffe, Pässe, Grate, Hügel, Berge, Seen, Täler, Wasserläufe, Gletscher, Moränen, Flüsse, Geröll, Sumpf

Vegetation: Wüste, Gras, Gebüsch, Bäume, Wald

Andere Merkmale: Pfade, Straßen, Hochspannungsleitungen, Pipelines, usw.

Knight entwirft auch ein Konzept zur Abbildung der Softwareinformationen auf die Metaphernelemente: Das gesamte Softwaresystem wird auf eine Landmasse abgebildet, einzelne Pakete auf Regionen der Landschaft, Klassen werden durch Täler und umgebende Gipfel gebildet, die ihrerseits die Methoden der Klasse darstellen können. Knight verfeinert ihr Konzept bis hin zu Bootsstegen und Touristinformationen als Repräsentationen.

Uns ist keine Realisierung einer Softwarevisualisierung bekannt, die eine solche explizite Abbildung vornimmt. Es existieren hingegen Systeme, die eine sehr einfache Landschaftsmetapher benutzen: Sie verwenden eingefärbte 3-D-Diagramme die einer Landschaft mit Bergen ähnlich sehen. Die Höhen und Tiefen dieser Visualisierungen ergeben sich durch die Häufung von Punkten der Ebene. Diese Punkte repräsentieren Elemente der Quelldaten, deren Koordinate aus der Ähnlichkeit untereinander berechnet wurde.

Bei [103] findet sich ein solches einfaches Beispiel der Landschaftsmetapher im Bezug auf ein Softwaresystem. Dort werden die Aufrufbeziehungen eines Softwaresystems von Nokia durch ein entsprechend eingefärbtes Höhenfeld visualisiert.

7.4.2 Repräsentationen der Stadtmetapher

Das Konzept der Stadtmetapher enthält sehr komplexe Grafikelemente und ermöglicht eine große Zahl von unterschiedlichen Repräsentationen in der Visualisierung.

Das Besondere an Stadtmetaphern ist, dass sie sehr komplexe Grafikelemente zulassen und so viele Repräsentationen in der Visualisierung kombinieren können.

Beispiele für Metaphernelemente der Stadtmetapher sind: Kontinente, Länder, Städte, Stadtbezirke, verschiedene Gebäude, Gebäudeelemente, Straßen, Wege, Teiche, usw.

[56] stellt für die Verwendung der Stadtmetapher ein sehr detailliertes aber auch beliebiges Konzept auf. Diese Abbildung ist im Softwarevisualisierungssystem „Software World“ realisiert worden: Das gesamte Softwaresystem wird durch die Welt dargestellt, Pakete werden auf Länder abgebildet und Städte auf Dateien. Städte enthalten ein Rathaus mit Dateiinformationen und einen Flughafen zur Navigation, d.h. zum Erreichen von anderen Städten. Städte werden durch eine Stadtmauer umschlossen. Klassen werden durch Stadtbezirke dargestellt. Sie sind gegeneinander durch Zäune abgegrenzt und enthalten jeweils einen Park, auf dem Vererbungsbeziehungen, implementierte Schnittstellen und Imports angegeben werden. Monumente innerhalb des Bezirks enthalten Informationen über Klassenvariablen. Methoden einer Klasse werden durch Gebäude repräsentiert. Die Parameter einer Methode werden durch Notausgänge des entsprechenden Gebäudes dargestellt. Der Name der Methode steht an einem Namensschild am Haupteingang und die geltenden Zugriffsbestimmungen auf diese Methode werden durch die Farbe des Gebäudes abgebildet. Im Gebäude werden außerdem Informationen zu den lokalen Variablen abgelegt.

Abbildung 30: Stadtmetapher mit eingebetteten Geschäftsinformationen. (Quelle: [PanasBerryganGrundy2003].)

Ein ähnliches Konzept haben [78] entworfen: Pakete des Softwaresystems werden durch Städte dargestellt, Klassen durch Gebäude. Deren Höhe entspricht der Anzahl der Quelltextzeilen der Klasse. Die Bebauungsdichte weist auf die Kopplung zwischen den Softwarekomponenten hin. Panas u. a. verwenden Straßen als bidirektionale und Flüsse als unidirektionale Kommunikationsverbindungen zwischen Paketen.

Neben die Abbildung der statischen Softwareinformationen stellen Panas u. a. auch dynamische Elemente. So sollen fahrende Autos oder Schiffe den Kommunikationsfluss im laufenden Programm repräsentieren. Wolken verdecken die Teile der dargestellten Welt, die aktuell nicht von Interesse sind.

Zusätzlich binden Panas u. a. Softwareprozessinformationen ein: Einfärbungen von Komponenten bilden die Aufgabenverteilung bei der Entwicklung der Komponenten ab. Flammen über Gebäuden deuten auf oft benutzte Programmteile hin während Blitze auf Klassen hinweisen, die häufigen Änderungen unterworfen sind. Abbildung 30 stellt eine Konzeptstudie für die Abbildung von Prozessinformationen dar.

7.4.3 Zweckmäßigkeit von Landschafts- und Stadtmetaphern

Softwarevisualisierungssysteme bilden Softwareproduktinformationen und Softwareprozessinformationen ab. Softwareproduktinformationen sind statische oder dynamische Informationen, die aus dem Quelltext abgeleitet oder bei dessen Abwicklung gesammelt werden. Softwareprozessinformationen beziehen sich auf den Entwicklungs- oder Betriebsprozess eines Softwaresystems. Hinzu kommen Softwareproduktmetriken bzw. Softwareprozessmetriken, die aus den anderen Softwareinformationen abgeleitet werden können. Beispiele für Softwareproduktinformationen sind:

- Klassen, Interfaces, Methoden, Attribute
- Ausführungspfade
- Anzahl der Codezeilen, Vererbungstiefe einer Klasse

Beispiele für Softwareprozessinformationen sind:

- Namen der an der Entwicklung beteiligten Mitarbeiter, Realisierungsfortschritt, zur Entwicklung benötigte Kosten
- durchschnittliche Nutzungszeit einer Komponente

Die abzubildenden Softwareinformationen werden je nach Zweck der Softwarevisualisierung ausgewählt. Diese werden dann jeweils auf die geeigneten Präsentationen abgebildet.

Auf relativ grober Ebene weisen Stadtmetaphern eine klare Struktur auf: Es gibt leicht identifizierbare Objekte wie Länder, Städte, Bezirke und Häuser und klar ersichtliche Beziehungen wie das Enthaltensein oder die Verbindung durch Straßen. Auf der Ebene dieser groben Repräsentationen wie Ländern, Städten, Bezirken und Häusern und deren Beziehungen untereinander können Stadtmetaphern deshalb sehr gut verwendet werden, um Strukturen aufzuzeigen.

Eine Landschaftsmetapher mit Landmassen, Bergen, Tälern und Seen ist ähnlich intuitiv und weist ebenfalls eine klare Struktur auf.

Sowohl bei der Stadt- als auch der Landschaftsmetapher sind die feingranularen Elemente

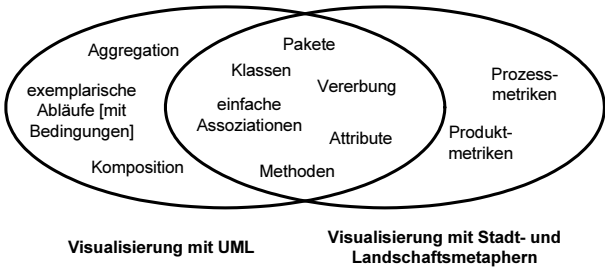


Abbildung 31: Vergleich der Softwareinformationen bei UML und Stadt- und Landschaftsmetaphern.

wie Fenster oder Türen bzw. Landungsstege oder Pässe weniger eindeutig. Verschiedene Menschen haben verschiedene Vorstellungen von den Objekten und deren Beziehungen zueinander. Hier existiert also ein Bruch im Verständnisprozess: Der Nutzer muss die feingranularen Repräsentationen explizit erlernen. Erschwerend ist hierbei jedoch die Komplexität und die Anzahl der Repräsentationen wie zum Beispiel von Häusern mit Fenstern, Türen, Dach, Schornsteinen, Fassade und Türschild. Sehr leicht wird dadurch die allgemeine Wahrnehmungsgrenze des Menschen von 5 bis 9 Entitäten – hier von Repräsentationstypen – überschritten.

Deshalb können mit Stadtmetaphern und komplexen Landschaftsmetaphern gut solche Softwareinformationen visualisiert werden, die ein grobes Verständnis über ein Softwaresystem vermitteln können. Entsprechende qualitative Informationen sind zum Beispiel Pakete, Klassen oder Methoden. Quantitative Informationen sind zum Beispiel Metriken wie die Anzahl der Zeilen oder die Anzahl der Anweisungen im Quelltext der Entitäten, die Anzahl der Methoden einer Klasse oder die Anzahl der Attribute einer Methode.

Die grobe Struktur der Stadt- und Landschaftsmetapher kann außerdem als Rahmen für die Visualisierung von anderen Softwareinformationen genutzt werden. So können wie bei [78] Softwareproduktinformationen auf die grobgranularen Repräsentationen projiziert werden.

Die beschriebenen 3-D-Diagramme sind einfache Ausprägungen von Landschaftsmetaphern. Deren Erstellung basiert auf der Abbildung von Ähnlichkeiten zwischen Entitäten. Deshalb ist diese Form der Landschaftsmetapher gut für die Darstellung der quantitativen Aspekte von Beziehungen zwischen Entitäten geeignet.

Die beschriebenen Landschafts- und Stadtmetaphernkonzepte basieren sehr auf der Darstellung von Entitäten und deren Beziehungen und sind deshalb gut für Programmiersprachen geeignet, die selbst solche Strukturen aufweisen – zum Beispiel für objektorientierte Programmiersprachen.

Stadt- und Landschaftsmetaphern können also besonders gut für Aufgaben verwendet werden, bei denen die Überblicksgewinnung über ein Softwaresystem im Vordergrund steht. Solche Aufgaben sind zum Beispiel die ersten Schritte beim Reverse-Engineering oder die Einarbeitung in den Entwicklungsprozess. Die Projektion von Softwareproduktinformationen kann für die Verfolgung des Entwicklungsstandes und damit für Management- und Controlling-Aufgaben von Interesse sein. In diesem Zusammenhang können Stadt- und Landschaftsmetaphern zur Visualisierung der Softwareevolution nützlich sein. Neben dem reinen Quelltext sind deshalb auch Prozessinformationen als Eingangsdaten für die Visualisierung sinnvoll.

Eine Softwarevisualisierung mit Stadt- und Landschaftsmetaphern auf dieser Ebene kann eine gute Ergänzung zu etablierten Softwarevisualisierungen wie zum Beispiel der Modellierungssprache UML[7] sein. Abbildung 31 zeigt einen Überblick über die durch UML und Stadt- und Landschaftsmetaphern visualisierten Softwareinformationen. Der wesentliche Mehrwert gegenüber UML besteht in der Abbildung von Metriken. Erst mit Metriken lassen sich bei einem Softwaresystem die Qualität und die Funktionalität bewerten.

7.5 Aspekte der Softwarevisualisierung bei Landschafts- und Stadtmetaphern

7.5.1 Automatische Anordnung der Repräsentationen

Der virtuelle Raum muss so erzeugt werden, dass er den menschlichen Anforderungen zur Wahrnehmung entspricht. Im Besonderen muss eine Desorientierung des Benutzers vermieden werden. Da Landschafts- und Stadtmetaphern 2,5-D-Darstellungen sind, kann das Problem der Anordnung der Repräsentationen auf die Anordnung in der Ebene, also im 2-D-Raum, beschränkt werden.

[55] beschreiben drei nützliche Verfahren zur Anordnung von Graphen:

Kräfteverfahren: Hierbei richten sich die Elemente durch gegenseitige Abstoßung aus. Die Abstoßungskräfte berechnen sich zum Beispiel aus der Kopplung von Klassen.

[7] Unified Modeling Language

Hierarchisches Verfahren: Hiermit können existierende Hierarchien, zum Beispiel Vererbungsbeziehungen, besser dargestellt werden. Das Verfahren verhindert, dass Verbindungen einen Knoten kreuzen und die Zahl der überkreuzten Kanten ist stark verringert.

Planarisierungsverfahren: Durch das Hinzufügen imaginärer Knoten wird die Topologie des Graphen verändert. Die Kanten werden anschließend orthogonal angeordnet und die Knoten werden so verteilt, dass der benötigte Platz möglichst gering ist. Das Planarisierungsverfahren führt zu einer ästhetischen Anordnung, die möglichst wenige kreuzende Verbindungen zwischen den Elementen enthält.

Besonders in Städten müssen die Häuser sinnvoll zu Bezirken zusammengefasst und arrangiert werden. [80] ordnet in seinem System „Zeugma“ die Funktionen eines Lisp-Programms mit Hilfe von so genannten Analogiekriterien nach funktionalen Einheiten: I/O-Funktionen, String-Funktionen, List-Funktionen, u. a. Dieses Vorgehen führt jedoch zu mehreren Städten für ein Programm – je nach bevorzugten Analogiekriterien.

Solche Ähnlichkeits-Zuordnungen sind auch bei den beschriebenen 3-D-Landschafts-Diagrammen nötig. Dort werden Beziehungs-Matrizen zwischen den Softwareinformationen gebildet und mit Hilfe verschiedener Algorithmen werden Positionen für ihre Repräsentation bestimmt. Boyack u. a. [10] nennen fünf solcher Algorithmen: Multidimensionales Skalieren, Hierarchisches Clustern, k-Durchschnitts-Algorithmen (k-means-algorithms), Principal Components-Analysis und selbstorganisierende Karten.

7.5.2 Interaktion, Navigation und Orientierung

Aufgaben, die der Benutzer eines Softwarevisualisierungssystems lösen will, sind zum Beispiel Exploration, Suche nach Informationen, Vergleichen von Informationen, Analyse von Elementen oder Beziehungen oder auch die Erzeugung neuer Informationen (nach [57]).

[72] nennen unter Anderem folgende Interaktionsanforderungen, die ein Softwarevisualisierungssystem erfüllen muss:

Überblick: Der Benutzer muss sich einen Überblick über den gesamten Datenraum verschaffen können.

Zoom: Der Benutzer muss in die Visualisierung hineinzoomen können. Dabei darf der globale Kontext nicht verloren gehen.

Filter: Der Benutzer muss unwichtige Teile ausblenden können. Dabei sollten Abstraktionsmechanismen verwendet werden, damit der globale Kontext nicht verloren geht.

Details: Der Benutzer muss Objekte oder Objektgruppen auswählen können, um Details anzuzeigen.

Beziehungen: Der Benutzer muss Beziehungen zwischen Elementen anzeigen können.

Verlauf: Die letzten Aktionen des Benutzers sollten vorgehalten werden und wiederholt bzw. rückgängig gemacht werden können.

Wichtiger Bestandteil der Interaktion sind die Navigation durch und die Orientierung in der Visualisierung. Navigation und Orientierung hängen eng mit der kognitiven Karte zusammen, die ein Benutzer von der Umwelt hat. Es gibt zwei Arten von kognitiven Karten: lineare Karten und räumliche Karten. Lineare Karten basieren auf den Bewegungen durch den Raum. Zumeist wird zuerst eine lineare Karte erstellt, die sich dann im Laufe der Zeit zu räumlichen Karten entwickeln (nach [57]). Verschiedene Interaktionsformen fördern den Aufbau dieser räumlichen Karten besser als andere. Zum Beispiel ist die aktive forschende Suche nach Informationen für den Lernprozess und die Verständnisgewinnung effektiver als eine durch das Visualisierungssystem angeleitete eher passive Erkundung.

Die Landschafts- und Stadtmetaphern enthalten bereits Navigationsformen, die auch vom Visualisierungssystem genutzt werden sollten. Diese sind zum Beispiel das Gehen, Fahren oder Fliegen. Wichtig ist die Effizienz der Navigation. Zum Beispiel kann es den Benutzer sehr behindern, wenn er von einem zum anderen Ende der Stadt „zu Fuß“ gehen muss. Zur Überwindung größerer Strecken ist auch das Teleportieren denkbar, wenngleich es nicht ursprüngliches Element des Metaphernkonzeptes ist.

Der Aufbau einer räumlichen Karte muss durch geeignete Orientierungsmechanismen unterstützt werden. Zum Beispiel erlaubt eine asymmetrische Anordnung der Repräsentationen in der Visualisierung eine bessere Orientierung.

[67] hat fünf wesentliche Gruppen von Orientierungselementen identifiziert (nach [47]):

Landmarken: Landmarken sind wiedererkennbare Objekte, die sich zum Beispiel durch einen hohen Grad an Sichtbarkeit auszeichnen. Sie geben Informationen über den aktuellen Standpunkt und die Richtung der Bewegung. Beispiele sind bekannte Gebäude oder Monumente wie sie

von [12] in der „Component City“ verwendet werden.

Bezirke: Bezirke sind Gebiete, die durch die Gleichartigkeit ihrer enthaltenen Objekte gebildet werden. Bezirke können zum Beispiel durch die gleiche geprägt sein.

Pfade: Die Hauptbewegungspfade einer Umgebung können zum Beispiel Hauptstraßen oder Fußwege sein.

Knoten: Knoten sind interessante Punkte entlang von Pfaden wie zum Beispiel Straßenkreuzungen oder Plätze.

Kanten: Kanten sind Begrenzungen von Bezirken oder Hindernisse für die Navigation. Dies sind zum Beispiel Zäune um einen Stadtbezirk, eine Autobahn um eine Stadt herum oder das Ufer, das eine Landmasse vom Meer trennt.

Diese Orientierungshilfen entstammen der Städteplanung und werden dort erfolgreich eingesetzt. Andere denkbare Orientierungshilfen sind die Integration von Karten, die den aktuellen Standpunkt in der Umgebung und eventuell die bereits gesehen Gebiete darstellen.

7.5.3 Abstraktion

Zur Unterstützung der Orientierung und der Verständnisgewinnung muss ein Softwarevisualisierungssystem verschiedene Abstraktionen anbieten können. Ziel der Abstraktion ist es aktuell unwichtige Informationen auszublenden. Hierarchische Abstraktionen enthalten verschiedene Detaillierungsgrade der Darstellung. Hierarchische Abstraktionsebenen sind zum Beispiel die Welt, das Land, die Stadt und ein Gebäude bzw. die entsprechend repräsentierten Softwareinformationen. Außerdem sind funktionale Abstraktionen denkbar, die bestimmte Repräsentationen, wie zum Beispiel alle Hilfsklassen des Systems, allein darstellen oder aber aus der Visualisierung ausschließen. Dazu muss das Softwarevisualisierungssystem die Selektion und das Einblenden bzw. Ausblenden von Objekten unterstützen. Bei jeglicher Abstraktion muss allerdings der Kontext erhalten bleiben. Für die Landschafts- und Stadtmetapher bedeutet das, die Einbettung in den durch Berge und Täler oder Städte und Länder gegebenen allgemeinen strukturellen Rahmen zu erhalten.

Die Landschafts- und Stadtmetapher bringen bereits geeignete Abstraktionsmittel mit. So sind Karten denkbar, die einen Überblick über die Landschaft oder die Stadt geben. Im Fall der Stadtmetapher sind globale Karten denkbar, die das gesamte Softwaresystem zeigen. Speziellere Karten können einen Überblick über einzelne Pakete oder auch Klassen des Softwaresystems geben.

Verschiedene Ansätze („Information Cube“ in [85], „Software Landscape“ in [6]) nutzen die Transparenz zur Realisierung von hierarchischer Abstraktion: Nahe Objekte werden als wichtig und aktuell interessant angenommen. Sie sind fast gänzlich transparent und geben den Blick frei auf enthaltene detaillierte Informationen. Hingegen werden weit vom Betrachter entfernte Objekte als uninteressant angenommen und trennen durch ihren hohen Opazitätsgrad die höheren Informationen von den niedrigeren, ohne jedoch die Informationen selbst zu verstecken.

7.5.4 Konfigurierbarkeit

Die Verwendung von Metaphern basiert auf der Nutzung und Unterstützung der menschlichen Fähigkeiten. Eine konsequente Fortführung dieses Anliegens erfordert, dass der Benutzer das Visualisierungssystem auch konfigurieren kann. Wenn mehrere Arten von Metaphern unterstützt werden, kann sich der Nutzer eine für ihn besonders ansprechende und auch aufgabengerechte Visualisierung auswählen. Auch die Konfigurierbarkeit der Menge der dargestellten Informationen kann sinnvoll sein: Nicht für jede Benutzergruppe und nicht für jede Aufgabe werden alle Daten über ein Softwaresystem benötigt. Es kann deshalb sinnvoll sein, Sichten zu definieren oder auch durch den Nutzer erstellen zu lassen. Dies erhöht das Verständnis und lenkt den Fokus auf die wichtigen Daten.

Bei der Verwendung von Sichten können unterschiedliche Softwareinformationen in verschiedenen Sichten auf die gleichen Repräsentationen abgebildet werden. Zwar muss der Benutzer in diesem Fall mehrere sichtenabhängige Abbildungen erlernen, andererseits erlaubt dies aber die Auswahl besonders geeigneter Repräsentationen.

7.5.5 Evolution

Die Konzepte der Landschafts- und Stadtmetaphern enthalten bereits die Idee der Entstehung einer Stadt bzw. der Besiedlung eines Landes. Deshalb kann Software-Evolution gut abgebildet werden. Mögliche Evolutionsvorgänge sind:

Entfernen von Softwarekomponenten: Das Entfernen einer Softwarekomponente, zum Beispiel einer Klasse, kann durch die Platzierung einer Abrisskrans an der jeweiligen Repräsentation angezeigt werden. Nach Entfernung der Softwarekomponente muss auch die Repräsentation entfernt werden.

Hinzufügen von Softwarekomponenten: Das Erzeugen von Softwarekomponenten kann durch das Aufstellen von Baugerüsten und Schildern angezeigt und durch den sukzessiven Aufbau der entsprechenden Repräsentation realisiert werden.

Einfache Quelltext-Änderungen: Änderungen am Quelltext sollten dem interessierten Benutzer der Visualisierung kundgetan werden. Dazu kann zum Beispiel ein Signal (ein Flagge oder ähnliches) an der Repräsentation angebracht werden, das nach einer angemessenen Zeit wieder entfernt wird.

Animationen: Durch die Animation der einzelnen Evolutionsschritte können zum Beispiel der Entwicklungs- oder der Wartungsprozess eines Softwaresystems dargestellt und so leicht nachverfolgbar gemacht werden.

Die konsistente Darstellung der Softwareevolution kann Probleme bei der Darstellung mit sich bringen, denn durch die Änderungen am Softwaresystem ergeben sich oft auch strukturelle Änderungen bei der Visualisierung. So kann zum Beispiel das Layout der Visualisierung ungültig werden, wenn sich die Funktionalität einer zugrunde liegenden Softwarekomponente ändert und somit deren bisherige Platzierung nicht mehr gültig ist. Bei der Erweiterung des Softwaresystems kann außerdem der benötigte Platz knapp werden. Dies kann nur durch eine vorausschauende Anordnung der Repräsentationen oder aber durch den teilweisen oder kompletten Neuaufbau der Visualisierung behoben werden. Im letzteren Fall muss dies dem Benutzer angezeigt werden, der daraufhin den entsprechenden Teil der Visualisierung erneut kennen lernen muss.

7.6 Einordnung und Bewertung

7.6.1 Forschung und Entwicklung

Verschiedene Forschungsgruppen beschäftigen sich mit der Nutzung von Landschafts- oder Stadtmetaphern zur Softwarevisualisierung.

An der University of Technology in Eindhoven, Niederlande, entwickelte die Gruppe um Telea ein Framework zur Softwarevisualisierung. Dabei haben sie auch eine einfache Landschaftsmetapher realisiert, die aus den Aufrufbeziehungen zwischen den Klassen des Softwaresystems eine gefärbte 3-D-Darstellung erzeugt. ([103])

An der Universität of Durham, England, beschäftigte sich die Gruppe um Munro und Knight mit einer sehr Quelltext-nahen Realisierung der Stadtmetapher für die Visualisierung von Java-Programmen. Die Arbeit mündete in den Prototypen „Software World“ ([56]). Sie wurde von Charter auf einer anderen Ebene wieder aufgenommen. Er wendete sich ab von der Repräsentation aller Elemente des Quelltextes hin zu einer Komponenten-basierten Betrachtung. Bisheriges Ergebnis ist der Prototyp „Component City“. ([12])

Ähnlich wie Knight arbeiten Balzer und Deussen an der Universität Konstanz, Deutschland: Ihre „Software Landscapes“ bilden sehr direkt die Strukturen und Beziehungen der Software ab. Sie haben das Konzept der Hierarchischen Netze entwickelt, mit dem sie die Beziehungen zwischen den Repräsentationen visuell ästhetisch durch explizite Verbindungen realisieren. Hierarchien werden durch die Schachtelung von Halbkugeln abgebildet. Zur Abstraktion wird eine entfernungsabhängige Transparenz verwendet. Das größte unter den mit ihrem System visualisierten Softwaresystemen ist das Programm Eclipse 2.02 mit einer Menge von mehr als 180.000 Quelltextzeilen. Der Mehrwert des Systems ist jedoch als gering einzuschätzen, da es bisher nur sehr wenige Softwareinformationen (Pakete, Klassen, Methoden, Attribute und Methodengröße) abzubilden scheint. ([6])

Auch Ploix an der Universität Paris verwendet eine Stadtmetapher zur Softwarevisualisierung. Diese ist ebenfalls sehr auf den Quelltext und besonders auf die enthaltenen Algorithmen ausgerichtet. ([80])

Wie schon erwähnt haben Panas u. a. eine Stadtmetapher entworfen, die neben den statischen und dynamischen Softwareproduktinformationen auch Softwareprozessinformationen abbildet. Dieses Konzept galt es im Rahmen einer allgemeinen Visualisierungs-Architektur erst noch zu implementieren. ([78])

Verwendung von Landschafts- und Stadtmetaphern zur Informationsvisualisierung

Auch außerhalb der Softwarevisualisierung scheint es nur wenige Anwendungen der Landschafts- oder Stadtdetapher zu geben.

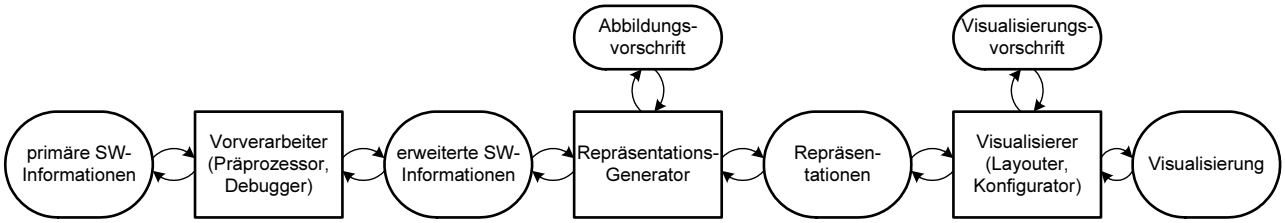


Abbildung 32: Allgemeiner Aufbau eines Softwarevisualisierungssystems.

Landschaftsmetaphern mit 3-D-Diagrammen finden sich beim kommerziellen System VxInsight, das von der Gruppe um Boyack an den Sandia National Laboratories, USA, entwickelt wurde. Es wurde zur Analyse von Literatur-, Patent- und Gen-Datenbanken eingesetzt. Einen frühen Beitrag zur Verwendung der Landschaftsmetapher für die visualle Analyse von Dokumentenbeständen stellt auch die Arbeit von Chalmers dar. [11]

Hartley und Churcher von der University of Canterbury, Neuseeland, benutzen die Landschaftsmetapher zur Visualisierung von Aufrufen der Universitäts-Internetseiten.

Der File System Navigator, der von der Silicon Graphics Inc., USA, entwickelt wurde, verwendet eine Stadtmetapher zur Visualisierung von Dateisystemen. [104]

Die Gruppe um Russo Dos Santos verwendet neben abstrakten Landschaftsmetaphern ebenfalls eine Stadtmetapher zur Darstellung von Daten über Netzwerke und Netzwerkdienste. [87]

7.6.2 Vergleich mit anderen Softwarevisualisierungen

Wie beschrieben existiert kein einheitliches Konzept zur Visualisierung von Softwareinformationen durch die Landschafts- oder die Stadtmetapher. Es gibt auch keine wirkliche Evaluierung der bestehenden Prototypen.

Die meisten Konzepte zur Softwarevisualisierung sind auf die Darstellung der statischen Systemarchitektur ausgerichtet. Adressiert werden dabei zumeist alle bekannten statischen Elemente wie Pakete, Klassen, Attribute und Methoden. Auch Beziehungen zwischen diesen werden abgebildet.

Anders als UML versuchen die meisten Systeme außerdem Softwareproduktmetriken abzubilden, was erheblich zum Verständnis eines Softwareproduktes beitragen kann, wie das durch bestehende 2-D-Softwarevisualisierungssystemen wie zum Beispiel [15] demonstriert wird.

Darüber hinaus äußern viele der Autoren den Wunsch in zukünftigen Versionen auch Elemente der dynamischen Systemarchitektur abzubilden. Auch die physische Architektur des Systems ließe sich zum Beispiel mit Landschaftsmetaphern abbilden (siehe [104]).

[78] gehen mit ihrem Konzept noch einen Schritt weiter und plant die Abbildung von Softwareprozessinformationen, die in UML und den meisten anderen Softwarevisualisierungen noch nicht berücksichtigt wurden.

7.6.3 Beispielhafter Ablauf des Visualisierungprozesses

Die untersuchten Arbeiten über Landschafts- und Stadtmetaphern zur Softwarevisualisierung sind zumeist sehr konzeptionell.

Deshalb betrachten wir hier am Beispiel der Stadtmetapher die Einbettung in einen allgemeinen Softwarevisualisierungsprozess wie er in Abbildung 32 dargestellt ist. Ausgangsdaten sind der Quelltext des Softwareproduktes und zusätzliche Softwareprodukt- oder Softwareprozessinformationen. In einem Vorverarbeitungsschritt können aus den statischen Softwareproduktinformationen mit Hilfe eines Debuggers dynamische Softwareproduktinformationen abgeleitet werden. Außerdem können zu diesen und bestehenden Softwareprozessinformationen entsprechende Metriken berechnet werden. All diese Informationen werden auf die Repräsentationen der Stadtmetapher – zum Beispiel Methoden auf Häuser – abgebildet. Diese Repräsentationen werden in der Visualisierungsstufe miteinander kombiniert – es wird zum Beispiel die Anordnung der Häuser zueinander aus den Aufrufbeziehungen untereinander abgeleitet. Ergebnis des Prozesses ist eine Visualisierung, die durch die jeweilige Hardware angezeigt werden muss.

Durch Abbildungsvorschriften wird festgelegt, wie die Abbildung der Softwareinformationen auf die Repräsentationen erfolgen soll oder welche Repräsentationen letztlich tatsächlich in der Visualisierung erscheinen sollen. Diese Vorschriften

sind nutzer- und aufgabenabhängig und sollten deshalb durch den Benutzer konfigurierbar sein.

Es muss außerdem berücksichtigt werden, dass die Verwendung der Metapher, also die Abbildungs- und Visualisierungsvorschriften, von der zugrunde liegenden Programmiersprache abhängig sind. So kann bei C++ zum Beispiel funktionaler Code auch außerhalb einer Klasse definiert werden – anders als bei Java.

Da Software potentiell unendlich komplex ist, sollte ein Softwarevisualisierungssystem diese Komplexität auch handhaben können und skalierbar sein. Bei Stadt- und Landschaftsmetaphern bedeutet dies im Besonderen einen schonenden Umgang mit den Grafikressourcen. Auf Modellseite muss die Zahl der darzustellenden Objekte begrenzt werden. Außerdem können aus der Computergrafik bekannte Level-of-Detail-Verfahren verwendet werden um die Grafikverarbeitung zu beschleunigen.

7.6.4 Diskussion

Intention der bisherigen Konzepte zu Landschafts- und Stadtmetaphern ist zum einen die Möglichkeit einen Zugang zu einem Softwaresystem zu bieten. Neben dieser groben Sicht wollen auch viele der Konzepte eine feingranulare Sicht auf die Software unterstützen. Zusätzlich wird auch die Möglichkeit zur Darstellung der Softwareevolution beschreiben.

Wir glauben, dass eine Stadtmetapher besonders für die grobe Erstanalyse eines Softwaresystems geeignet sein kann, da sie gut die groben Strukturen abbilden kann. Wir glauben weiterhin, dass es sehr schwer ist, feingranulare Softwareelemente und deren Beziehungen so durch Stadtmetaphern abzubilden, dass die gute Intuitivität der Metapher erhalten bleibt. Denn möchte ein mit der Softwareentwicklung vertrauter Nutzer wirklich in einer Stadt umherlaufen um die Informationen zu erhalten: Möchte er wirklich wie bei [56] ein Monument besteigen, um die Klassenvariablen einer Klasse zu erfahren?

Stadt- und Landschaftsmetaphern könnten aber für solche Nutzergruppen sinnvoll sein, die mit anderen Softwarebeschreibungen nicht vertraut sind und auch nicht an der feinen Softwarestruktur interessiert sind. Dies ist gerade der Ansatz den [78] vorstellen. Sie verwenden die Stadtmetapher als strukturellen Rahmen, in den Softwareprozessinformationen eingeblendet werden. Die kann zum Beispiel für das Management oder den Kunden interessant sein. Hier kann tatsächlich die Vertrautheit mit der Metapher genutzt werden um schnell einen allgemeinen Überblick über das System zu erhalten, wie zum Beispiel die Größe einer Komponente oder die voneinander abhängigen Komponenten.

Leider sind Prototypen der Systeme nicht verfügbar. Deshalb können wir die die bisherigen Ergebnisse zur Visualisierung der groben und feinen Softwareproduktinformationen oder der Softwareprozessinformationen nicht testen und beurteilen.

7.7 Zusammenfassung

Softwarevisualisierungssysteme mit Landschafts- und Stadtmetaphern versuchen die Vorteile von 3-D-Darstellungen auszunutzen und diese zugleich durch die Beschränkung auf das 2,5-dimensionale wirklich nutzbar zu machen. Wir haben eine Reihe von Arbeiten zur Umsetzung dieser Metaphern untersucht. Dabei ist festzustellen, dass Landschaftsmetaphern weniger Beachtung finden als Stadtmetaphern: Abgesehen von den vorgestellten 3-D-Diagrammen existiert keine Umsetzung der Landschaftsmetapher. Sowohl Softwareprozess- als auch Softwareproduktinformationen können auf eine Vielzahl von Repräsentationen abgebildet werden. Dies sind zum Beispiel Länder, Städte, Bezirke oder Häuser. Diese Repräsentationen sollten aufgaben- und nutzerorientiert in einer Reihe von Sichten visualisiert werden. Für die Nutzbarkeit des Systems sind zum einen eine gute Automatisierbarkeit und eine gute Skalierbarkeit auf der anderen Seite aber auch eine sehr gute Orientierungs- und Navigationsunterstützung für den Nutzer notwendig.

Die Forschung auf dem Gebiet der Softwarevisualisierung durch Landschafts- und Stadtmetaphern scheint nur sehr langsam voran zu gehen. Die Konzepte sind noch sehr allgemein und beliebig. Sie enthalten eine Menge von Wünschen, umgesetzt wurde aber zumeist nur ein sehr kleiner Teil. Dieser Eindruck wird dadurch verstärkt, dass keine der Prototypen-Implementierungen verfügbar sind und getestet und beurteilt werden könnten. Die Frage was denn genau eine Stadtmetapher oder eine Landschaftsmetapher ausmacht, ist nicht geklärt. Damit ist aber auch nicht geklärt welche Elemente des Metaphernkonzeptes wirklich genutzt werden können. Die Forschung scheint noch mit der Erörterung des Machbaren und noch nicht mit einer Umsetzung des Sinnvollen beschäftigt.

Kapitel 8

Visualisierung von Softwareevolution

Michael Schöbel

Kapitel 8: Visualisierung von Softwareevolution

Michael Schöbel

Zusammenfassung. Software verändert sich. Aus Nutzersicht zeigen sich Änderungen in verschiedenen Versionen mit unterschiedlichem Funktionsumfang. Aus Entwicklersicht unterliegen Quelltextdateien und weitere zum Programm gehörende Dokumente (z.B. Dokumentation und Spezifikation) einem Veränderungsprozess. Informationen über den Änderungsvorgang sind bei der Softwareentwicklung von wichtiger Bedeutung. Entsprechend aufbereitet können diese Informationen zu einer Verbesserung verschiedener Aspekte des Entwicklungsprozess führen. In der vorliegenden Arbeit werden Visualisierungskonzepte für Evolutionsvorgänge in Software erläutert und gezeigt, wie diese den Entwickler bei der Arbeit unterstützen können. Abschließend wird am Beispiel von *Augur* ein Werkzeug zur Evolutionsvisualisierung vorgestellt.

8.1 Einleitung

Ein Softwaresystem wird durch zahlreiche Dateien beschrieben. Dazu gehören einerseits die Quelltextdateien und andererseits weitere Dateien, welche Dokumentation, Anforderungsspezifikationen, Testprotokolle und ähnliche Informationen enthalten. Im gesamten Nutzungszeitraum der Software unterliegen diese Dateien verschiedenen Änderungen: Es werden Fehler gefunden und beseitigt, neue Funktionalität wird spezifiziert und implementiert, schon fertiggestellte Teile werden optimiert, usw. Dies führt zu einer stetigen Veränderung der Software.

Größere Softwareprojekte werden von einem Team von Entwicklern durchgeführt. Auch dieses Team unterliegt Änderungen: im Laufe der Zeit kommen neue Entwickler hinzu und andere beteiligen sich nicht mehr an der Entwicklung.

Daraus lassen sich zwei Komplexitätsbereiche der Softwareentwicklung ableiten [34]. Ein Bereich ist die zu erstellende Software (Artefakte). Dabei muss die Struktur der Software (also Klassen, Interfaces, Aufrufbeziehungen u.ä.) betrachtet werden. Der zweite Bereich ist der Software-Entwicklungsprozess (Aktivitäten). Dort spielen unter anderem Anforderungen, verschiedene Softwarereleases und Vorgehensmodelle eine Rolle. Beide Bereiche können getrennt betrachtet werden. Werden nur Klassen und Interfaces analysiert, können zum Beispiel Abhängigkeiten zwischen diesen erkannt werden. Betrachtet man nur die kürzlich durchgeführten Aktivitäten einzelner Entwickler kann erkannt werden, welche Personen für welche Teilbereiche zuständig sind. Jedoch nur durch die Verknüpfung beider Aspekte kann beispielsweise erkannt werden, welche Klassen von der kürzlich von Entwickler X durchgeführten Änderung betroffen sind. Es zeigt sich also, dass durch eine getrennte Betrachtung beider Bereiche wichtige Informationen übersehen werden können.

Softwareevolution verbindet beide Komplexitätsbereiche: Eine Klasse (= Artefakt) wird im Laufe der Zeit von einem oder mehreren Entwicklern geändert (= Aktivität). Eine Visualisierung der Änderungsprozesse ermöglicht es abstrakte Sachverhalte „auf einen Blick“ zu erkennen: Welcher Entwickler arbeitet an welchen Dateien? Wo wurden die letzten Änderungen durchgeführt? Dadurch kann eine verbesserte Koordination der Entwickler erreicht werden. Außerdem können neu hinzukommende Entwickler einen schnellen Überblick über das Gesamtsystem erhalten und dabei neben den vorhandenen Klassen auch sehen, welche Entwickler diese Klassen zu welchem Zeitpunkt implementiert haben.

Im nachfolgenden Abschnitt wird erläutert, welche Daten die Grundlage für die Erfassung der Softwareevolution bilden und wie diese gewonnen werden können. Im anschließenden Hauptteil werden verschiedene Visualisierungskonzepte für die Darstellung von Änderungsprozessen in Softwaresystemen vorgestellt. Abschließend wird Augur vorgestellt, ein Visualisierungswerkzeug, welches einige der Konzepte realisiert.

8.2 Datenbasis

Für eine angemessene Darstellung von Softwareevolutionsprozessen müssen in einem ersten Schritt die relevanten Rohdaten identifiziert werden.

8.2.1 Versionsverwaltung

In den meisten mittleren bis großen Softwareprojekten werden Versionsverwaltungssysteme einge-

setzt, welche die Projektdateien in einem zentralen Repository verwalten. Dadurch wird es für Entwickler möglich an verschiedenen Orten auf die Quellcode-Dateien zuzugreifen („Check-Out“), diese zu modifizieren und anschließend wieder im Repository abzulegen („Check-In“). Dabei überwacht die Versionsverwaltung problematische Operationen, wie das gleichzeitige Bearbeiten einer Datei.

Außerdem wird jeder „Check-In“-Vorgang protokolliert und die an einer Datei durchgeführten Änderungen gespeichert. Dies ermöglicht es jede ältere Versionen eine bestimmten Datei zu rekonstruieren und dadurch die Entwicklung einer Datei über einen bestimmten Zeitraum zu verfolgen. Abbildung 2.1 zeigt, welche Informationen zu jedem „Check-In“ erfasst werden: der Autor, der Zeitpunkt, ein Kommentar (eingegeben vom Autor) und die Änderungsinformationen zu jeder betroffenen Datei. Der Kommentar sollte dabei den Grund für die Änderung und die Änderung selbst genauer beschreiben.

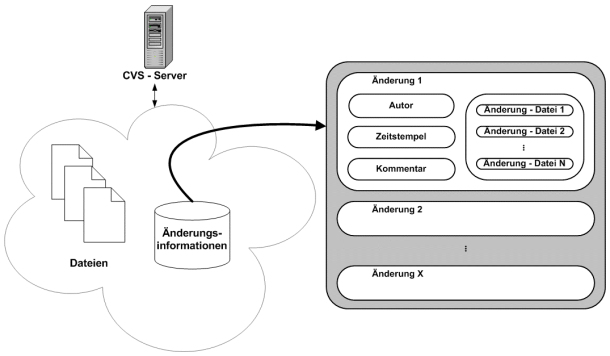


Abb. 2.1: In einem Versionsverwaltungssystem abgelegte Informationen

Die erfassten Informationen stellen die Grundlage für die Beschreibung von Änderungsprozessen dar. Dabei ist besonders hervorzuheben, dass diese Daten implizit erfasst werden und durch die Verwendung eines Versionierungssystems „nebenbei“ anfallen. Ein Visualisierungswerkzeug kann die Schnittstelle zum Versionsverwaltungssystem verwenden und ist entsprechend einfach zu installieren. Weiterhin kann es sofort ohne Vorbereitung verwendet werden und erfordert keine zusätzliche Aktivität durch die Entwickler.

8.2.2 Externe Informationen

Die Entwicklung eines Softwaresystems zeigt sich jedoch nicht nur allein an den Quellcodedateien. Zusätzliche relevante Informationen werden auch außerhalb der Versionsverwaltungssysteme abgelegt: Testprotokolle, Anforderungsspezifikationen, Informationen über Aufwand der durchgeführten Änderungen, usw. Weiterhin können zu den verschiedenen Revisionen einer Quellcode-Datei die bekannten Softwaremaße (Anzahl der Codezeilen, Anzahl der Methoden, ...) ermittelt werden und damit deren Entwicklung über die Zeit erfasst werden.

Um diese (Versionssystem-)externen Informationen zu erfassen, ist zusätzlicher Aufwand notwendig. Für die Verknüpfung der extern abgelegten Informationen mit den Quellcode-Dateien wäre es beispielsweise denkbar, dass im vom Entwickler eingegebenen Kommentar ein Hinweis auf ein externes Dokument enthalten ist. Die Entwicklung der Softwaremaße kann auch von dem Visualisierungswerkzeug (unter Umständen in einem Präprozessing Schritt) erstellt werden, indem alle abrufbaren Revisionen einer Datei mit einem externen Werkzeug vermessen werden.

8.2.3 Verarbeitung

Die ermittelten Basisdatenelemente können in zwei Gruppen eingeteilt werden.

- Voneinander unabhängige Entitäten. Zum Beispiel einzelne Autoren, Zeitpunkte oder Dateien.
- Kontrollierbare Variablen. Zum Beispiel der Änderungsumfang in Zeilen, Aufwand in Tagen oder verschiedene Softwaremetriken.

Eine Visualisierung der Daten soll den Zusammenhang zwischen den Entitäten und Variablen verdeutlichen. Dabei werden abhängig von der konkreten Aufgabenstellung Entitäten und Variablen ausgewählt und auf verschiedene Darstellungsformen abgebildet. Im folgenden Abschnitt werden verschiedene Abbildungsmöglichkeiten vorgestellt.

8.3 Visualisierungskonzepte

Die Herstellung einer geeigneten Darstellung für komplexe Zusammenhänge ist teilweise ein kreativ-künstlerischer Prozess. Dementsprechend viele verschiedene Möglichkeiten gibt es. Dennoch lassen sich objektive Bewertungskriterien definieren, welche einen Vergleich verschiedener Ansätze ermöglichen.

8.3.1 Anforderungen

Eine Darstellung von Softwareänderungsprozessen sollte es dem Betrachter ermöglichen eine oder mehrere folgender Fragestellungen bezüglich verschiedener Programmelemente zu beantworten [107]:

- Was ist seit ... passiert?
- Wer ist dafür verantwortlich?
- Wo hat sich etwas geändert?
- Wann wurde die Änderung durchgeführt?
- Warum wurde die Änderung durchgeführt?
- Wie wurde eine Datei geändert?
- Wie hat sich eine Datei im Laufe der Zeit verändert?

In eingeschränktem Umfang könnten auch zukünftig geplante Änderungen relevant sein und visualisiert werden. Problematisch ist dabei, dass der zukünftig erreichte Zustand nicht aus den schon vorhandenen Daten abgeleitet werden kann und damit spekulativ ist.

Eine Visualisierung soll den Betrachter dabei unterstützen, neue Einsichten bezüglich einer bestimmten Aufgabe zu gewinnen. Die Zweckmäßigkeit einer Darstellung ergibt sich damit aus dieser Aufgabenstellung. Zusätzlich sind die vorhandenen Navigationsmittel wichtige Bewertungskriterien: Wie werden verschiedene Datensätze verknüpft? Können irrelevante Daten ausgefiltert werden? Wie kann man sich zwischen den Daten bewegen?

Neben den genannten Punkten ist die Benutzerunterstützung eine weiteres Unterscheidungskriterium: Muss der Benutzer selbständig die relevanten Informationen in der Darstellung *erkennen*, oder ist das Werkzeug in der Lage dem Benutzer die Besonderheiten zu *zeigen*?

Die folgenden Ansätze werden auch bezüglich dieser Fragen betrachtet.

8.3.2 Klassische Ansätze

Zu den klassischen Ansätzen werden Visualisierungskonzepte gezählt, die nicht speziell für die Darstellung von Evolutionsprozessen entworfen wurden, die aber in anderen Bereichen verbreitet sind. Im Allgemeinen muss eine klassische Darstellung um die zeitliche Dimension ergänzt werden.

8.3.2.1. Matrix-Ansichten

Eine zweidimensionale Matrix-Ansicht stellt Werte in Abhängigkeit von zwei Dimensionen, deren Entitäten auf orthogonalen Achsen abgebildet sind, dar. An den Schnittpunkten können verschiedene Maße visualisiert werden. Dabei sind durch die Gestaltung der Darstellung mehrere Werte kodierbar. Beispielsweise könnten auf einer Achse die beteiligten Entwickler, auf der anderen Achse die zum Projekt gehörenden Dateien und an den Schnittpunkten Quadrate abgebildet werden, deren Seitenlänge die Zahl der Codezeilen des Entwicklers an der entsprechenden Datei darstellt. Zusätzlich kann durch die Farbe des Quadrates der Zeitpunkt der letzten Änderung visualisiert werden. Der Betrachter kann dadurch auf einen Blick erkennen, welche Entwickler maßgeblich an welchen Dateien beteiligt sind (Größe der Quadrate) und in welchen Dateien zuletzt Änderungen vorgenommen wurde (Farbe).

Matrix-Ansicht erlauben die Darstellung von sehr vielen Zellen. Die Interaktion ist dabei für den Benutzer intuitiv verständlich, dass heißt zoomen und drehen der Darstellung sind einfach realisierbar. Im Falle einer dreidimensionalen Darstellung kann jedoch die Übersicht verloren gehen, da im Vordergrund stehende Elemente dahinterliegende verdecken könnten.

8.3.2.2. Graphen

Graphen sind eine naheliegende Darstellung von Beziehungen zwischen Entitäten. Für jede Entität wird dabei ein Knoten erzeugt. Besteht zwischen zwei Entitäten eine Beziehung wird eine Kante eingezeichnet. Darüberhinaus gibt es zahlreiche Gestaltungsmöglichkeiten für die Darstellung verschiedener Maße: Eigenschaften der Entitäten können durch Knotengröße, -farbe oder -form verdeutlicht werden. Darstellungsparameter für Kanten (Breite, Farbe, Länge) können Eigenschaften der entsprechenden Beziehungen kodieren.

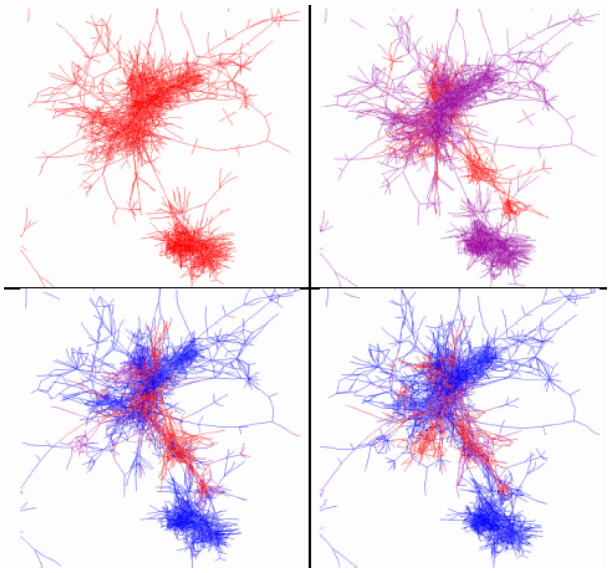


Abb. 3.1: Visualisierung eines Aufrufgraphes (Knoten = Klassen)

Die Abbildung 3.1 (aus [17]) zeigt eine Graphendarstellung eines Aufrufgraphes für ein Softwaresystem. Jeder Knoten entspricht dabei einer Klasse, eine Kante zwischen zwei Knoten zeigt, das

eine der beiden Klassen eine Methode der anderen Klasse aufruft. Die Färbung der Darstellung stellt eine Besonderheit dar und verdeutlicht einen Aspekt der Softwareevolution: Jede der vier Grafiken stellt das Gesamtsystem zu unterschiedlichen Zeitpunkten dar. Dabei visualisiert die Farbe eines Knoten oder einer Kante den Zeitpunkt der letzten Änderung, kürzlich modifizierte Elemente werden rot dargestellt und dann mit zunehmenden Alter blau gefärbt. Auf diese Weise können auch sehr große Softwaresysteme kompakt dargestellt werden und der Betrachter kann die kürzlich geänderten Klassen sofort erkennen.

Theoretisch lassen sich also komplexe Zusammenhänge durch eine Graphenstruktur veranschaulichen. Problematisch ist jedoch die Darstellung: Es existiert kein Verfahren zur optimalen Darstellung beliebiger Graphen. Durch verschiedene heuristische Verfahren (z.B. Federalgorithmen) können auch umfangreiche Graphen visualisiert werden, die Darstellungsqualität hängt aber stark von der Struktur des Graphen ab und ist damit nicht allgemein bewertbar. Die Repräsentation von Beziehungen zwischen den Knoten als Kanten ist ebenfalls ein problematischer Punkt: Durch Überschneidungen der Kanten geht die Übersicht schnell verloren. Sind dann noch verschiedene Maße als grafische Darstellungsattribute der Kanten kodiert, wirkt die Darstellung sehr schnell chaotisch.

Trotz der Schwächen, hat die Graphendarstellung auch Vorteile gegenüber den anderen Verfahren: Man kann eine kompakte, optisch leicht erfassbare Darstellung des Gesamtsystems generieren. Durch Interaktion (verschieben von Knoten, rein- und rauszoomen) kann der Benutzer relativ einfach einzelne Knoten auswählen. Diese beiden Möglichkeiten zusammen führen zu einer hilfreichen Darstellung: In der (unter Umständen unübersichtlichen) Darstellung des Gesamtsystems können Besonderheiten hervorgehoben werden, zum Beispiel durch eine entsprechende Farbwahl der Knoten. Dabei kann der Benutzer „auf einen Blick“ die Knoten erkennen, die er sich genauer ansehen sollte. Durch Auswahl (und damit eventuell verbundenen Wechsel der Visualisierungsart) kann er die hervorgehobenen Knoten genauer analysieren.

Als Übersichtsdarstellung und Ausgangspunkt für die weitere Analyse ist die Graphendarstellung demnach auch bei unübersichtlichen Datenmengen geeignet. Bei geeigneter Wahl der Darstellungsattribute lassen sich mit Graphen auch verschiedene Aspekte von Änderungsprozessen visualisieren.

8.3.2.3. Codezeilenbasierte Darstellung

Die Codezeilenbasierte Darstellung ist ein realtiv gut erforschtes Verfahren [28] und wird allgemein zur Visualisierung quelltextbezogener Information verwendet. Dabei wird jede Zeile einer Quellcode-Datei als eine horizontale Pixelline repräsentiert. Die Länger der Linie ist proportional zur Länge der Zeile in der Datei.

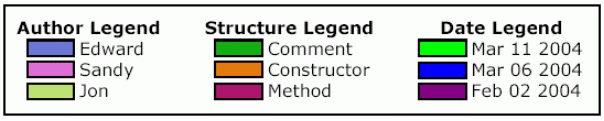


Abb. 3.2: Farbliche Kodierung

Farblich können jetzt verschiedene Informationen pro Codezeile kodiert werden (Abbildung 3.2), zum Beispiel der Entwickler der diese Zeile zuletzt geändert hat, der Typ der Codezeile (Kommentar, Schleife, ...) oder Zeitpunkt der letzten Änderung. Durch die Darstellung der Quellcodezeilen können diese Informationen dann visuell den einzelnen Codezeilen zugeordnet werden.

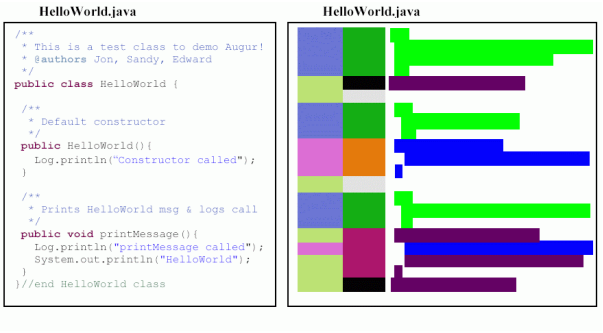


Abb. 3.3: Zuordnung zu Codezeile

Werden Informationen über den Änderungsprozess in einem Softwaresystem codezeilenbasiert dargestellt, ergibt sich vor allem für die beteiligten Entwickler ein Informationsgewinn: Ein Programmierer erkennt „seine“ Codestruktur wieder und kann beispielsweise auf einen Blick sehen, welche Kollegen an von ihm geschriebenen Zeilen Modifikationen durchgeführt haben. Auch für einen Gesamtüberblick ist die codezeilenbasierte Visualisierung bei einer entsprechenden Farbwahl geeignet, so kann sofort erkannt werden, an welchen Dateien kürzlich gearbeitet wurde, oder welche Entwickler an welchen Dateien gearbeitet haben.

8.3.3 Evolutionsmatrix

Die Evolutionsmatrix (aus [61]) wurde im Unterschied zu den klassischen Ansätzen speziell für die Darstellung von Softwareevolution entwickelt.

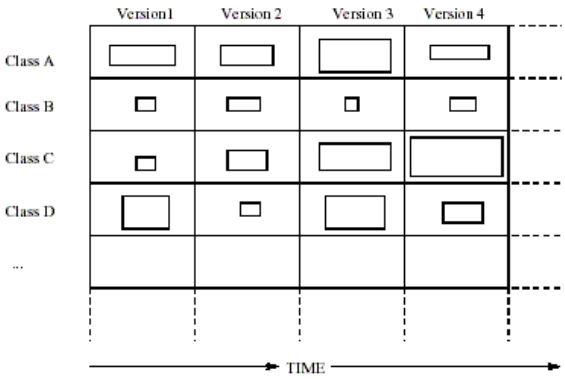


Abb. 3.4: Darstellung von Klassen in unterschiedlichen Versionen

Jede Zeile der Matrix wird einer Klasse des zu visualisierenden Systems zugeordnet. Jede Spalte entspricht einer (Release-)Version des Gesamtsystems. In den Zellen der Matrix werden zwei unterschiedliche Metriken zu der entsprechenden Klasse in der entsprechenden Version durch Rechtecke dargestellt: Die Breite und die Höhe der Rechtecke entsprechen den Werten je eines bestimmten Maßes. Zum Beispiel kann die Breite die Anzahl der Methoden der Klassen und die Höhe die Anzahl der Klassenvariablen darstellen.

Die Klassen (= Zeilen der Matrix) werden nach zwei Kriterien sortiert: (1) Zuerst werden die Klassen nach Versionen geordnet. Klassen die in einer früheren Version existieren stehen weiter oben. (2) Innerhalb einer Versionsgruppe (= Klassen einer bestimmten Version) werden die Klassen alphabetisch sortiert.

Mit Hilfe der Evolutionsmatrix-Darstellung können jetzt Analysen auf zwei Ebenen durchgeführt werden: auf der Ebene des Gesamtsystems und für einzelne Klasse. In Abb. 3.5 ist beispielhaft ein Überblick über ein Gesamtsystem dargestellt.

Ohne die konkrete Form der Rechtecke zu betrachten, können schon auf hoher Ebene Informationen über das Softwaresystem erkannt werden.

- Wachstums- und Stagnationsphasen des Systems. Durch den Vergleich von zwei Spalten können neu hinzugekommene Klassen erkannt werden.
- Gesamtgröße. Die letzte Spalte der Matrix enthält alle in der aktuellen Version existierenden Klassen.

Für die Analyse einer einzelnen Klasse wird nur die Zeile der entsprechenden Klasse betrachtet.

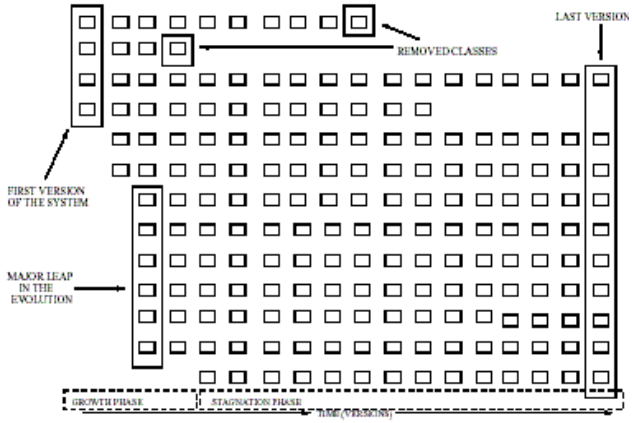


Abb. 3.5: Gesamtsystemdarstellung

Dabei kann die Formänderung des dargestellten Rechteckes interpretiert werden. In [61] werden optische Muster zur Identifikation von Entwicklungstypen vorgestellt. Die Begriffe sind dabei zu einem großen Teil aus der Astronomie entlehnt. In Abbildung 3.6 sind drei Muster dargestellt.

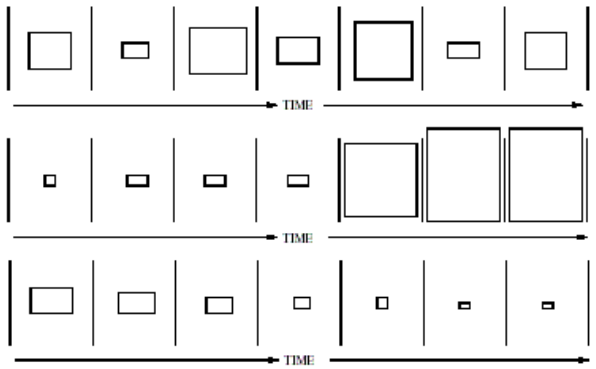


Abb. 3.6: Entwicklungstypen: (von oben) Pulsar, Supernova, Weißer Zwerg

Aus den erkannten Mustern können Informationen über die Klassen abgeleitet werden.

Pulsare sind Klassen, die zyklisch Wachsen und Schrumpfen. Dies könnte zum Beispiel daraufhin deuten, das es sich um sehr wichtige Klassen handelt, da sie in fast jeder Version des Gesamtsystems geändert werden.

Supernovas sind Klassen, die plötzlich in ihrer Komplexität anwachsen. Grund könnte sein, dass die Klasse anfangs nur ein leeres Interface definierte und anschließend fertig implementiert wurde.

Weiße Zwerge werden Klassen bezeichnet, die erst ihre Komplexität verlieren und anschließend auf geringem Niveau verharren. Dabei könnte es sich um Klassen handeln die „tot“ sind und nicht mehr benötigt werden.

Eintagsfliegen sind Klassen die nur in genau einem Release des Systems vorhanden waren. Mit diesen Klassen muss man sich nicht weiter beschäftigen.

Persistente Klassen sind Klassen, die seit der ersten Version des Systems enthalten sind. Das könnte zum Beispiel bedeuten, dass diese Klasse ein besonders gutes Design aufweist und daher nie geändert wurde. Andererseits wäre es auch möglich, dass die Klasse noch nie verwendet wurde.

Prinzipiell ist die Darstellung als Evolutionsmatrix gut geeignet um sich einen Überblick über das visualisierte System zu verschaffen. Die aktuellen Klassen (letzte Spalte) und deren Entwicklung können gut erkannt werden. Zusätzliche Information über den Autor der Klasse könnte noch als Farbe des Rechteckes visualisiert werden. Probleme ergeben sich aus dem Zeitbezug: Die Unterteilung nach Software-Releases ist eher zu grob, so dass Entwicklungen zwischen zwei Releasezeitpunkten verloren gehen. Weiterhin geht das Sortierungssystem der Klassen davon aus, dass sich der Klassenname nicht ändert. Bei einer Umbenennung "'springt"' die Klasse an das Ende der Liste und könnte daher für eine neu hinzugefügte Klasse gehalten werden.

Das größte Problem ergibt sich jedoch aus der Interpretation durch den Betrachter. Die Bedeutung der „optischen“ Muster kann nur zusammen mit der Rechteckdefinition (welche Maße werden für Breite und Höhe verwendet) erklärt werden. Eine feste Interpretationsvorschrift kann es daher nicht geben.

8.3.4 Revisionstower

Revisionstower sind ein Visualisierungskonzept, welches in [101] vorgestellt wurde. Die grundsätzliche Idee ist, die Entwicklung zweier zusammengehöriger Dateien gleichzeitig zu visualisieren. Solche Dateipaare können sich aus verschiedenen Gründen ergeben. Handelt es sich beispielsweise um ein C++-Projekt, ist es sinnvoll Header- und Implementierungsdatei gegenüberzustellen. Bei Java könnte eine Interface-Definition zusammen mit einer implementierenden Klasse betrachtet werden.

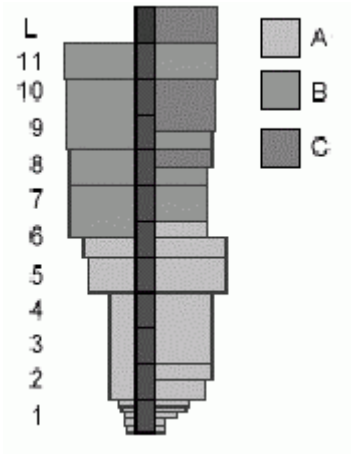


Abb. 3.7: Gegenüberstellung von zwei Dateien

In Abbildung 3.7 ist ein Revisiontower dargestellt. Die mittlere Achse stellt dabei den Zeitbezug her: Für jeden Software-Release wird ein gleichgroßer Abschnitt reserviert. Es ist ebenfalls möglich, die Größe der Abschnitte nach der wirklich verbrauchten Zeit zu skalieren, so dass für längere Entwicklungszyklen auch mehr Raum für die Visualisierung zur Verfügung steht. Rechts und links von der mittleren Achse wird je eine Datei dargestellt. Dabei entspricht jedes Rechteck einer Datei-Revision. Die Länge des Rechteckes symbolisiert die Gesamtzahl der Codezeilen der Datei, die Farbe kodiert den Autor, der für die entsprechende Revision verantwortlich ist. Es wären aber auch andere Informationsabbildungen denkbar.

Ein Softwaresystem wird also als Sammlung von Revisionstürmen visualisiert werden. Durch Ein- und Ausblenden von Dateirevisionen (= Rechtecke im Turm) oder ganzen Türmen, kann der Zustand des Gesamtsystems kontinuierlich dargestellt werden. Durch Analyse einzelner Türme können verschiedene Informationen gewonnen werden:

- Der Entwickler X ändert nie Headerdateien.
- Der Entwickler Y hat seit Release R keine Änderungen durchgeführt.
- Die Datei D wächst ständig.

Bei sehr großen Systemen kann die Turmdarstellung unübersichtlich wirken: Es gibt sehr viele Türme, unter Umständen sind viele Türme leer oder enthalten nur auf einer Seite Rechtecke, da es keine „Partner“-Datei gibt. Die Beziehung zwischen den einzelnen Türmen und deren Revisions-Rechtecken ist nicht erkennbar. Die zeitliche Auflösung der Darstellung ist genauer als bei der Darstellung als Evolutionsmatrix.

8.4 Tool: AUGUR

Abschließend soll ein Beispielsystem evaluiert werden, welches einige der vorgestellten Visualisierungskonzepte verwendet und zur Veranschaulichung von Softwareevolutionsprozesse verwendet werden kann.

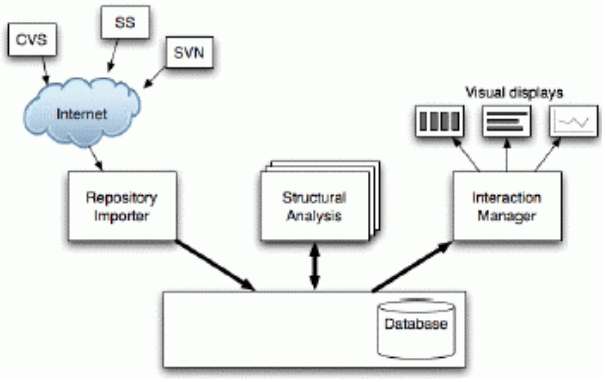


Abb. 4.1: Augur-Architektur

Abbildung 4.1 zeigt die Augur-Architektur: Angeschlossen an ein Versionierungssystem werden die Dateien aus dem Repository heruntergeladen und lokal abgespeichert. Zusätzlich zu den reinen Quellcode-Dateien werden die Datei Annotationen und die Log-Daten abgelegt, um die Änderung einer bestimmten Datei verfolgen zu können. Je nach verwendeter Programmiersprache, kann Augur die Programmstruktur analysieren und beispielsweise den Typ einer einzelnen Quellcodezeile (Kommentar, Konstruktor, ...) feststellen. Nach dieser Datenaufbereitung, kann der Nutzer mit verschiedenen Sichten die gesammelten Informationen analysieren.

Abb. 4.2: Primärsicht: Codezeilenbasiert

Als primäre Sicht verwendet Augur eine codezeilenbasierte Darstellung. Konfigurierbar ist, welche Information farblich pro Codezeile kodiert ist: das Alter der Codezeile, der Entwickler der die letzte Änderung durchgeführt hat oder der Typ der Codezeile. Letzteres kann momentan nur bei Java-Projekten ausgewählt werden, da für andere Programmiersprachen kein Parser implementiert ist der diese Information erfassen könnte.

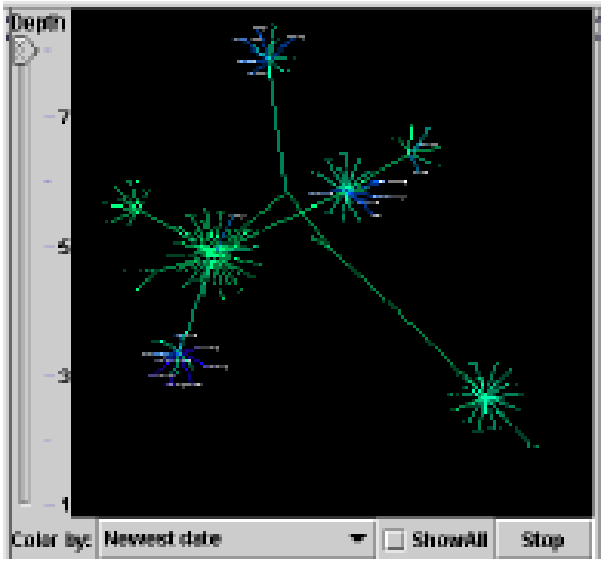


Abb. 4.3: Sekundärsicht: Graphendarstellung

Sekundär können zahlreiche andere Sichten ausgewählt werden. Zum Beispiel eine textuelle Zusammenfassung der Entwicklungsgeschichte (Welches ist die älteste Datei? Wer ist der aktivste Entwickler? ...) oder eine Graphendarstellung der Arbeitsgruppen, das heißt alle Entwickler sind als Knoten dargestellt und genau dann verbunden, wenn sie an einer Quellcodedatei gemeinsam gearbeitet haben. Die Linienbreite der Kanten kodiert dabei die Anzahl der gemeinsam bearbeiteten Dateien. Weiterhin ist eine Kalenderansicht wählbar, in der die Entwicklungsaktivität bestimmten Wochentagen zugeordnet wird. Speziell für Java-Projekte existieren noch Sichten, die Teile der statischen Programmstruktur (zum Beispiel Interface-Implementierung oder Vererbungsbeziehungen) visualisieren.

Augur befindet sich in einer frühen Entwicklungsstufe und hat daher noch Schwächen bei der Stabilität und Performance der zahlreichen Funktionen. Dennoch demonstriert es sehr gut, wie verschiedene Visualisierungskonzepte verwendet werden können, um unterschiedliche Informationen über die Evolution eines Systems greifbar zu machen.

8.5 Einordnung und Bewertung

Softwareevolution wird meistens in Bezug auf die Quellcode-Dateien visualisiert. Dies ergibt sich aus der Anschlussmöglichkeit an ein Versionierungssystem, welches auch nur Dateien und keine Klassen unterscheiden kann. Es gibt aber auch Möglichkeiten Elemente der statischen oder dynamischen Systemstruktur zusammen mit Evolutionsinformationen zu visualisieren (zum Beispiel: Xia [107]). Auch Augur erlaubt eine einfache Analyse der statischen Struktur von Java-Programmen.

Allgemein haben Werkzeuge zur Visualisierung von Softwareevolution daher folgende Eigenschaften:

- Sie dienen zur nachträglichen beziehungsweise begleitenden Analyse von Teilen oder des gesamten Systems.
- Es werden alle Programmiersprachen unterstützt, sofern die Dateien in einem Versionierungssystem enthalten sind. Es ist möglich, dass ein bestimmtes Werkzeug für bestimmte Programmiersprachen besondere Analysemittel zur Verfügung stellt und daher besser für die Analyse geeignet ist.
- Besondere Programmierkonventionen müssen im Allgemeinen nicht eingehalten werden, da die Informationen implizit aus dem Versionierungssystem gewonnen werden können.
- Die visualisierbare Systemgröße hängt von dem konkreten Werkzeug ab. Prinzipiell sind jedoch Projekte beliebiger Größe darstellbar.

Das vorgestellte Werkzeug Augur wurde von J. E. Froehlich und Prof. J. P. Dourish an der University of California, Irvine entwickelt. Augur ist in Java geschrieben und daher auf Plattformen mit einer Java Runtime ausführbar. Augur ist frei nutzbare Software [4] und befindet sich noch in der Testphase. In der aktuellen Version 0.9.3 können beliebige mit CVS verwaltete Projekte visualisiert werden. Zur Verfügung stehen verschiedene Sichten, so dass der Nutzer sich eine geeignete Darstellung zusammenstellen kann. Daher ist eine sofortige Visualisierung möglich, ohne aufwändige manuelle Vorbereitungen. Zur Skalierbakeit ist zu sagen, dass eine ausreichende Hauptspeichergröße und eine gute Grafikkarte entscheidend sind: ein Rechner mit 512MByte RAM und 32MB-Grafikkarte ist ausreichend um das mittelgroße Projekt VRS [106] mit ca. 1.200 Dateien und 300.000 Codezeilen zu visualisieren. Die Entwickler von Augur haben ihr System vorwiegend mit kleineren Open-Source Projekten getestet. Augur wird aktuell weiterentwickelt und soll in folgenden Versionen weitere Funktionen zur Verfügung stellen und dann zum Beispiel auch zusammen mit anderen Versionierungssystemen (Subversion, Visual Source Safe) verwendet werden können. Außerdem sollen zusätzliche Layout-Algorithmen für die Primärsicht und weitere Werkzeuge zur Analyse von Abhängigkeiten zwischen den Dateien (Call-Graph, Social Networks) implementiert werden.

8.6 Ausblick

Die Evolution von Softwaresystemen ist ein wichtiger Aspekt im Softwareentwicklungsprozess. Vor allem für die arbeitsteilige Bearbeitung von Softwareprojekten ist die Koordination von entscheidender Bedeutung. Eine hilfreiche Darstellung der Änderungsvorgänge (zum Beispiel: Wer arbeitet an welchen Dateien?) kann die Koordination verbessern. Darüber hinaus kann durch die Darstellung der Veränderung verschiedener Qualitätsmaße eine Verbesserung der Gesamtgüte des Softwareproduktes erreicht werden, da Problemfelder besser und früher erkannt werden können.

Ein entsprechendes Visualisierungswerkzeug sollte den Benutzer bei der Analyse unterstützen und sowohl einen Gesamtüberblick ermöglichen, als auch Details darstellen können. Eine Verknüpfung verschiedener Darstellungstypen führt zu einer besseren Interaktivität. Die Art der Darstellung ist dabei sehr stark abhängig von der Art der zu beantwortenden Fragen.

Für die Weiterentwicklung der vorhandenen Ansätze können folgende Punkte eine Rolle spielen:

Skalierbarkeit Wie kann der Speicherbedarf verringert werden? Wie kann der Benutzer bei der Analyse sehr großer Systeme unterstützt werden? Alle der vorgestellten Visualisierungskonzepte haben Probleme mit sehr großen Datenmengen.

Datenkonsistenz Ist es möglich, Datei- oder Klassenumbenennung automatisch zu erkennen? Die Änderungsinformationen sind in einer Versionsverwaltung dateibezogen gespeichert. Wird eine Datei oder Klasse umbenannt, so wäre es ideal, wenn ein Werkzeug diesen Vorgang erkennt und damit die Änderungsinformationen entsprechend zuordnen kann. Ein ähnliches Problem ergibt sich bei unterschiedlichen Namen (im Versionierungssystem) des gleichen Entwicklers.

Kapitel 9

Ergebnisse und Ausblick

Johannes Bohnet

Kapitel 9: Ergebnisse und Ausblick

Johannes Bohnet

Zusammenfassung. Die Seminarbeiträge behandeln fünf für die Softwarevisualisierung wichtige Themenkomplexe: a) Quellcodenahe Visualisierungstechniken, b) Verwendung von Softwaremetriken, c) Visualisierung von Softwareevolution, d) Metaphernverwendung bei der Abbildung der Softwareartefakte in den Darstellungsraum und e) Exploration des Informationsraums. Die Analyse dieser Themenkomplexe ermöglicht zum einen die Identifizierung von Problembereichen als mögliche Ursachen für eine mangelnde Akzeptanz von Softwarevisualisierungs-Werkzeugen. Eine wichtige Rolle hierbei spielen die drei Aspekte Skalierbarkeit, Automatisierbarkeit und Exploration. Zum anderen können sinnvolle Anwendungsbereiche für die Visualisierung komplexer, objektorientierter Softwaresysteme ausgewiesen werden. Zu nennen sind diesbezüglich a) die metrikbasierte Visualisierung statischer Softwareaspekte, b) die Visualisierung der Systemstruktur auf Basis dynamisch erzeugter Informationen und c) die Visualisierung der Softwareevolution in Verbindung mit Prozessinformationen.

9.1 Ergebnisse

Visualisierungstechniken können helfen, ein Verständnis über die Struktur, das Verhalten und die zeitliche Entwicklung eines komplexen Softwaresystems, sowie der damit verbundenen Entwicklungs- und Managementprozesse zu entwickeln. In diesem Report fokussieren wir auf Visualisierungstechniken, die auf der Implementierung eines Softwaresystems basieren und nicht auf expliziten Systemmodellen, da während des Lebenszyklus des Systems Inkonsistenzen zwischen den expliziten Modellen und der Implementierung entstehen können. Von einer gültigen Systembeschreibung durch explizite Modelle kann daher nicht ausgegangen werden (siehe Abschnitt 1.1.1).

Aus dem Bereich des Forward-Engineerings existieren Werkzeuge, mit deren Hilfe sich automatisiert aus der Implementierung explizite Modelle im Sinne der UML [75] erzeugen lassen (z. B. Borland Together [9] oder IBM Rational [45]). Für die Analyse komplexer Softwaresysteme sind automatisch generierte Visualisierungen dieser Werkzeuge aufgrund der großen Datenmengen jedoch nur bedingt von Nutzen:

Zum einen können sich Schwierigkeiten bei der Anordnung der Informationsrepräsentationen im Darstellungsraum ergeben. Für große Graphen sind angepasste Konzepte notwendig, um Darstellungen zu erzeugen, die trotz vieler knotenverbindenden Kanten übersichtlich sind.

Zum anderen müssen für die Exploration der dargestellten Informationen Mechanismen zur Navigation im Darstellungsraum bereit gestellt werden. Für große Informationsvolumen ist es notwendig, diese speziell anzupassen bzw. zu entwickeln, damit sie sinnvoll nutzbar sind.

9.1.1 Akzeptanz von Softwarevisualisierungs-Werkzeugen

Die Tatsache, dass sich bisher kein implementierungsbasiertes Softwarevisualisierungs-Werkzeug als ein Standard für Aufgaben im Entwicklungsprozess komplexer Softwaresysteme hat etablieren können, verdeutlicht, dass bei Softwarevisualisierungs-Werkzeugen vielschichtige Akzeptanzprobleme existieren. Der hier vorliegende Report zeigt mögliche Ursachen für eine geringe Nutzbarkeit von Visualisierungs-Werkzeugen auf:

- Die Extraktion der darzustellenden Informationen erfordert manuellen Aufwand.
- Die Datenextraktion, die Layout-Verfahren oder die Explorationsmechanismen sind nicht skalierbar und damit für komplexe Softwaresysteme ungeeignet.
- Die Exploration der dargestellten Informationen ist nicht intuitiv. Es werden Standard-Navigationstechniken verwendet, die nicht auf den Darstellungsraum spezialisiert sind und die Nutzer ungenügend bei der Exploration unterstützen.
- Eine Visualisierungstechnik wird für die Beantwortung unterschiedlicher softwaretechnischer Fragestellungen verwendet und hat aufgrund ihrer Universalität nicht die jeweils optimale verständnisfördernde Wirkung.
- Das Softwarevisualisierungs-Werkzeug ist nicht in eine Entwicklungsumgebung integriert. Die parallele Verwendung von Entwicklungsumgebung und externen Werkzeugen bedeutet einen Mehraufwand für Softwareentwickler.

Bei der Entwicklung eines Visualisierungs-Werkzeug, das von einer breiten Anwenderschicht genutzt werden soll, muss besonderer Wert auf die genannten Aspekte gelegt werden.

Bisher wurden wenige Versuche unternommen, Softwarevisualisierung empirisch zu evaluieren. Als wichtige Untersuchungen sind die Arbeiten von Bassil und Keller [7] und von Koschke [58] zu nennen. Es zeigt sich, dass von den Nutzern neben anderen Aspekten insbesondere geeignete Explorations- und Navigationsmechanismen in hierarchischen Darstellungen gewünscht werden.

9.1.2 Untersuchte Konzepte

Die im Rahmen des Reports *Konzepte der Softwarevisualisierung für komplexe, objektorientierte Softwaresysteme* erfolgte Analyse existierender Konzepte identifiziert für die Softwarevisualisierung wichtige Fragestellungen:

- Wie hoch sollte der Abstraktionsgrad einer Visualisierung für verschiedene softwaretechnische Fragestellungen sein? Für welche Fragestellungen sind insbesondere quellcodezeilenbasierte Visualisierungstechniken sinnvoll?
- Inwiefern können Softwaremetriken für die Softwarevisualisierung verwendet werden?
- Welche zusätzlichen Informationen neben Informationen über die vorliegende Implementierung des Softwaresystems können automatisiert gewonnen werden? Wie können Management-Aufgaben dadurch unterstützt werden?
- Wie sollte der Darstellungsraum beschaffen sein?
- Welche Mechanismen zur Exploration der dargestellten Informationen sollte ein Werkzeug den Nutzern zur Verfügung stellen?

Im folgenden werden fünf für die Softwarevisualisierung wichtige Themenbereiche besprochen, die die oben genannten Fragestellungen adressieren:

1) Quellcodedarstellung:
   Für Aufgaben im Softwareentwicklungsprozess, die nicht ein Strukturverständnis über das Softwaresystem verlangen, sondern deren primärer Untersuchungsgegenstand einzelne Quellcodezeilen sind, werden spezielle Visualisierungstechniken benötigt. Diese sollten einerseits erlauben, große Mengen an Quellcode darzustellen, andererseits sollte die dem Softwareentwickler bekannte visuelle Struktur erhalten bleiben.
2) Softwaremetriken:
   Informationen über statische Aspekte von Softwaresystemen im Sinne der UML [75] benennen Softwareartefakte und ihre Beziehungen untereinander. Mit Hilfe von Metriken werden Eigenschaften der Artefakte und deren Beziehungen quantifiziert und so wird eine Bewertungsgrundlage geschafft.
3) Softwareevolution:
   Softwaresysteme müssen im Laufe ihres Lebenszyklus stetig durch Softwareentwickler weiterentwickelt werden. Das Verständnis über die Entwicklung des Systems und den Aktivitäten der Softwareentwickler kann helfen, frühzeitig problematische Tendenzen zu erkennen und vorausschauend zu reagieren.
4) Darstellungsraum:
   Die Wahl des Darstellungsraums ist maßgeblich für die Softwarevisualisierungstechnik. Wichtige Aspekte hierbei sind die Dimension des Raumes, das Metaphernsystem zur Abbildung der Softwareartefakte und die Mechanismen zur Navigation im Darstellungsraum.
5) Exploration:
   Bei der Visualisierung von komplexen Softwaresystemen wird ein sehr großes Informationsvolumen dargestellt. Es müssen geeignete Navigationsmechanismen für die Exploration des dargestellten Informationsraums zur Verfügung gestellt werden.

9.1.3 Quellcodezeilenbasierte Softwarevisualisierung

Für Aufgabenbereiche, in denen nicht das Verständnis über die Struktur von Softwaresystemen im Vordergrund steht, sondern sich Fragestellungen direkt auf den Quellcode beziehen, können quellcodezeilenbasierte Visualisierungstechniken sinnvoll eingesetzt werden. Diese Techniken abstrahieren in der Darstellung nur wenig vom Quellcode. Sie stellen ihn in einer kompakten, miniaturisierten Weise dar. Auf diese Weise kann eine erweiterte Menge an Quellcode dargestellt werden, wobei die visuelle Struktur des Quellcodes erhalten bleibt, so dass sie von den Softwareentwicklern wiedererkannt wird. Informationen über die Quellcodezeilen werden als Farbwerte kodiert.

Auf dieser Technik basierende Softwarevisualisierungs-Werkzeuge, die Informationen über statische Aspekte von Softwaresystemen darstellen, sind SeeSoft [28], AspectBrowser [2], Aspect Mining Tool [3] und sv3D [69]. Bei Augur [4] gehen zusätzlich noch Informationen über die Evolution des System mit in die Visualisierung ein.

Interessante Anwendungsbereiche für quellcodezeilenbasierte Visualisierungstechniken finden sich bei der Untersuchung von Informationen über dynamische Aspekte eines Softwaresystems. Ex-

emplarisch zu nennen sind hierbei die Fehlersuche oder die Prüfung von Codeüberdeckung.

Informationen über dynamische Aspekte eines Softwaresystems lassen durch Instrumentierung des Quellcodes oder des Bytecodes eines Softwaresystems erhalten. Im ersten Fall werden entweder automatisiert oder manuell durch den Entwickler Anweisungen in den Quellcode eingefügt, die während der Laufzeit Ausgaben über den Zustand des Programms erzeugen. Im Fall der Bytecode-Instrumentierung werden im ausführbaren Programm Anweisungen identifiziert, die z. B. Aufrufe von Funktionen oder das Reservieren von Speicher einleiten. An diesen Stellen wird zusätzlicher Bytecode eingefügt, der während der Laufzeit Ausgaben über den Systemzustand erzeugt. Über im ausführbaren Programm enthaltene Debug-Informationen lassen sich die beobachteten Ereignisse auf den Quellcode zurückführen. Bei beiden Arten der Instrumentierung wertet das Visualisierungssystem die Ausgaben über den Systemzustand entweder direkt aus oder sie werden erst in einer Log-Datei gesammelt, um anschließend visualisiert zu werden. Bei in Java implementierten Softwaresystemen können zusätzlich über die Reflection-API [102] Laufzeitinformationen abgefragt werden.

Auf dem Markt erhältliche Profiler-Werkzeuge, die eine Instrumentierung der Software automatisiert durchführen, sind z. B. IBM Rational Purify [46] oder Compuware DevPartner [18]. Ihre Anwendung zielt hauptsächlich auf die Performanz-Analyse des Systems oder auf das Auffinden von Fehlern bei der Speicherverwaltung.

Als Softwarevisualisierungs-Werkzeuge, die Laufzeitinformationen auf Basis von quellcodezeilenbasierte Visualisierungstechniken darstellen, sind zu nennen Tarantula [42], Gammatella [54], Bloom-Bee/Hive [83] und Cleanscape Testwise [14].

9.1.4 Metrikbasierte Softwarevisualisierung

Zur Darstellung von komplexen Informationsräumen, die keine inhärente hierarchische Struktur aufweisen, können Metriken[8] über Aspekte des Softwaresystems zur Herausbildung einer Struktur im Darstellungsraum genutzt werden. Unter Ausnutzung der ausgeprägten menschlichen Fähigkeit zur Mustererkennung können Nutzer Softwareartefakte extrahieren, die eine Besonderheit bezüglich den verwendeten Metriken besitzen. Der metrikbasierte Ansatz zur Visualisierung von Software ist eine Art Filtermechanismus, um für Nutzer möglicherweise interessante Elemente des Softwaresystems zu identifizieren. Nutzer erhalten Einstiegspunkte in das System, von denen aus sie ihr Verständnis über das System verfeinern können. Für diesen nachfolgenden Schritt müssen sie allerdings auf andere Techniken zur Verständnisgewinnung zurückgreifen. Im einfachsten Fall wäre dies die Betrachtung des Quellcodes.

Softwaremetriken geben quantitativ Aufschluss über Eigenschaften von Softwaresystemen. Eine umfassende Zusammenstellung von objektorientierten Softwaremetriken ist in [66] gegeben. Im folgenden werden in der Softwarevisualisierung häufig verwendete Metriken genannt. Sie lassen sich untergliedern anhand des Eigenschaftsbereiches, den sie charakterisieren.

Metriken, die die **Komplexität** eines Artefakts quantifizieren sind:

- #[9]Quellcodezeilen, mit denen das Artefakt beschrieben wird. (LOC)
- # Methoden einer Klasse (NOM)
- # Attribute einer Klasse (NOA)
- Tiefe des Vererbungsbaumes (DIT)
- # direkter Subklassen einer Klasse (NOC)

Metriken über die **Kohäsion** von Artefakten geben an, wie eng Teilelemente eines Artefakts miteinander in Verbindung stehen:

- Methodenpaare einer Klasse können eine gemeinsame Instanzvariable benutzen. LCOM gibt die Anzahl solcher Methodenpaare einer Klasse an abzüglich der Anzahl der Methodenpaare, die keine gemeinsame Instanzvariable benutzen.

Metriken über die **Kopplung** charakterisieren die Stärke von Verbindungen zwischen Artefakten:

- # Klassen, die mit einer Klasse über Methoden- oder Attributaufrufe mit einander in Beziehung stehen.

Metrikbasierte Softwarevisualisierungstechniken lassen die quantitativen Aussagen von Metriken in Layout-Verfahren einfließen. In sehr einfacher Form wird dies z. B. bei dem Visualisierungswerkzeug CodeCrawler [15] im 2 dimensionalen Darstellungsraum umgesetzt. Der Wert einer Metrik für ein Softwareartefakt wird je nach Darstellungsmodus auf die Position des Darstellungselementes entlang einer Achse, auf seine Größe oder Farbe oder auf die Stärke einer Verbindungslinie

[8] Mit dem Begriff Metrik sind im weiteren Produktmetriken über objektorientierte Softwaresysteme gemeint.

[9] Das Symbol # ist eine Ersetzung des Ausdrucks „Anzahl von".

abgebildet. Verschiedene Darstellungsmodi, bei denen die Abbildungsvorschriften für Metriken auf Eigenschaften der Darstellungselemente vorkonfiguriert sind, zielen darauf, einzelne Repräsentationen von Artefakten optisch hervortreten zu lassen, die eine softwaretechnische Besonderheit aufweisen. CodeCrawler bietet einige für das Reverse-Engineering nützliche Darstellungsmodi, die einen Überblick über die Komplexität des Softwaresystems geben oder Einstiegspunkte in das System identifizieren, bei denen eine weitere Analyse sinnvoll sein kann.

Die genannten Layout-Verfahren sind bezüglich der Anordnung der Elemente im Darstellungsraum stark eingeschränkt. Metriken werden primär auf Eigenschaften der Darstellungselemente wie der Größe, Form oder Farbe abgebildet. Die Positionierung der Darstellungselemente ist meist auf ein Raster oder durch eine einfache, diskrete Strukturierungsregel festgelegt. Im Gegensatz dazu erlauben Graph-Layout-Verfahren, die auf der Simulation eines physikalischen Kräftesystems beruhen, Softwareartefakte und ihre Beziehungen untereinander in eine räumliche Struktur zu überführen, deren Ausgestaltung maßgeblich von einer Metrik abhängt. Die menschliche Fähigkeit zur Mustererkennung erlaubt es den Anwendern effizient, in der räumlichen Struktur Gruppierungen und Partitionierungen zu identifizieren, die möglicherweise eine softwaretechnische Relevanz besitzen. kräftebasierte Darstellungstechniken werden u.a. in den Softwarevisualisierungs-Konzepten von SHriMP/Creole [97], CrocoCosmos [19] und JST [48] verwendet.

Die besprochenen objektorientierten Produktmetriken werden auf Basis der Quellcodebeschreibung eines Softwaresystems gebildet. Insbesondere ist daher die Aussagekraft von den Kopplungsmetriken eingeschränkt, welche die Kopplung zwischen Softwareartefakten aufgrund ihrer Aufrufbeziehungen charakterisieren. Polymorphie als Eigenschaft objektorientierter Programme oder die Verwendung von Verzweigungs-Anweisungen erschwert den Verständnisprozess, da Darstellungen, die nur auf Basis des Quellcodes erstellt werden, nicht das wirkliche Verhalten des Systems zur Laufzeit abbilden können.

9.1.5 Softwareevolution

Softwaresysteme unterliegen während ihres Lebenszyklus einem stetigen Wandel aufgrund von Fehlerkorrekturen, veränderten Benutzeranforderungen oder einer modifizierten Systemumgebung. Ein Verständnis über die zeitliche Entwicklung eines Softwaresystems kann helfen, frühzeitig entstehende Probleme zu identifizieren und darauf vorausschauend zu reagieren. Visualisierungen können dabei zwei Komplexitätsbereiche adressieren. Zum einen lässt sich die zeitliche Entwicklung der Softwareartefakte selbst darstellen. Zum anderen können Informationen über die Softwareentwickler im Mittelpunkt der Visualisierung stehen.

Für die sinnvolle Verwendung eines Visualisierungs-Werkzeugs ist es notwendig, dass die dargestellten Informationen weitgehend automatisiert aus dem Softwaresystem extrahiert werden. Es bietet sich an, dazu Versionsverwaltungssysteme zu nutzen, da deren Verwendung bei komplexen Softwaresystemen mit einer größeren Anzahl von Entwicklern ohnehin unerlässlich sind. Die Informationen, die mit Hilfe eines Versionsverwaltungssystems zur Verfügung stehen, sind die einzelnen Änderungen im Quellcode verknüpft mit dem Namen des Autors der Änderung, einem Zeitstempel und einem Kommentar des Autors bezüglich der Änderung.

Um frühzeitig problematische softwaretechnische Tendenzen in einem System zu identifizieren, ist es sinnvoll Momentaufnahmen des Softwaresystems visuell gegenüber zustellen. Dazu können Visualisierungen zur Analyse statischer Informationen über Softwaresysteme mit Informationen über die zeitliche Entwicklung angereichert werden. Das Visualisierungs-Werkzeug GEVOL [17] berücksichtigt bei der Graph-Darstellung von Klassen eines Softwaresystems, z. B. bezüglich ihrer Aufrufstruktur, den Zeitpunkt der letzten Änderung an den einzelnen Klassen. Ähnlich lässt sich bei SHriMP/Xia [107] die von SHriMP/Creole bekannte Darstellung statischer Aspekte eines Softwaresystems mit Evolutionsinformationen erweitern. Als kommerzielles Werkzeug, mit dem Veränderungen von Abhängigkeiten einzelner Systemkomponenten und Metriken über diese in einer Graph-Darstellung visualisiert werden können, ist Headway reView [40] zu nennen.

Andere Konzepte visualisieren Softwareartefakte durch eine Matrix-Darstellung, in der eine Raumdimension der zeitlichen Entwicklung der Artefakte entspricht. Softwaremetriken beeinflussen die Geometrie der Darstellungselemente. Exemplarisch hierfür sind die Konzepte Evolutionsmatrix [61] und Revisiontowers [101] zu nennen.

Das Visualisierungs-Werkzeug SeeSoft [92] nutzt quellcodezeilenbasierte Visualisierungstechniken, um Quellcode entsprechend des Alters der einzelnen Zeilen farbig markiert darzustellen. Es können Bereiche im Quellcode identifiziert werden, die über die Zeit unverändert bleiben, was auf nicht mehr genutzte Systemteile hinweisen kann. Augur [4] erweitert das Konzept von SeeSoft um die Darstellung von Informationen über die an der

Entwicklung des Softwaresystems beteiligten Programmierer. Für Management-Aufgaben können diese Darstellungen sinnvoll sein, da auf diese Weise die Zuständigkeitsbereiche im Quellcode und die Aktivitäten der einzelnen Programmierer ersichtlich werden.

9.1.6 Eigenschaften des Darstellungsraums

Die meisten Konzepte zur Softwarevisualisierung bilden Softwareartefakte in einen 2 dimensionalen Darstellungsraum ab. Darstellungen mit 2 Raumdimensionen haben gegenüber der 3 dimensionalen Darstellung den Vorteil, dass die für die Informationsexploration nötigen Navigationstechniken wesentlich einfacher sinnvoll realisiert werden können. Im Vergleich zu 2D Umgebungen besteht in 3D Umgebungen eine ungleich größere Gefahr, dass Nutzer während der Navigation ihre Orientierung verlieren. Dass heißt, sie verlieren das Gefühl für ihre Position und Ausrichtung im Darstellungsraum. Bei der Entwicklung von Navigationstechniken für 3D Umgebungen muss daher ein besonderes Gewicht auf die Orientierungsunterstützung der Nutzer gelegt werden.

Andererseits bieten 3D Darstellungen gegenüber 2D Darstellungen einige Vorteile. Die Dichte der darstellbaren Informationen ist größer. Darstellungselemente können zusätzlich entlang der dritten Dimension positioniert werden. Zudem können geometrisch komplexere Darstellungselemente für die Repräsentation der Softwareartefakte gewählt werden. Eine Informationsüberflutung des Nutzers sollte dabei jedoch vermieden werden.

Für die Visualisierung großer Informationsvolumen bietet es sich an, für die Darstellung einen 3 dimensionalen Darstellungsraum zu wählen, bei dem eine Raumrichtung speziell ausgezeichnet ist. Diese 2½ dimensionalen Darstellungsräume vereinen die Vorteile sowohl der 2 dimensionalen als auch der 3 dimensionalen Darstellungen. Zum einen ermöglichen sie eine hohe Informationsdichte. Zum anderen vermindert die Existenz einer ausgezeichneten Achse im Raum die Gefahr der Nutzerdesorientierung. Visualisierungskonzepte basierend auf Landschafts- oder Stadtmetaphern sind Beispiele für die Nutzung eines 2½D Darstellungsraums.

Im Kontext der Softwarevisualisierung können Landschaftsmetaphern für Visualisierungstechniken verwendet werden, die auf ein Verständnis der Beziehungsstruktur von Softwareartefakten zielen. Ähnlich wie bei der graphbasierten Darstellung (siehe Abschnitt 9.1.4) können mit Hilfe von Metriken Softwareartefakten Positionen auf einer abstrakten Landschaft zugeordnet werden. Die Morphologie der Landschaft ergibt sich aus der Häufung von zugeordneten Positionen. Eine solche Landschaft könnte mit Darstellungselementen angereichert werden, die Softwareartefakte explizit repräsentieren, ähnlich wie dies bei der Verwendung von Stadtmetaphern geschieht.

Solche erweiterten Landschafts- oder Stadtdarstellungen eignen sich, um hierarchische Strukturen darzustellen, da die Anwender eine ihnen aus der realen Welt bekannten Hierarchie auf das Softwaresystem übertragen können. Ein Beispiel hierfür ist die Hierarchiefolge Kontinent, Land, Stadt und Haus. Idealerweise werden die Darstellungselemente so gewählt, dass ihre Semantik von Navigationstechniken genutzt werden kann. Das Selektieren einer Straße, welche z. B. eine Aufrufbeziehung zwischen zwei Instanzen einer Klasse repräsentiert, könnte das Navigieren zur jeweils anderen Instanz einleiten.

Landschafts- und Stadtmetaphern eignen sich nur zur Darstellung von Informationen, die stark von den durch den Quellcode eines Softwaresystems beschriebenen Artefakten abstrahieren. Zur Darstellung von Details auf einer niedrigen Abstraktionsebene sind sie nicht geeignet. So wirkt es für Softwareentwickler eher behindernd, wenn z. B. die Implementierungsdetails einer Klasse in dem Inneren eines Gebäudes abgelesen werden müssen, wie es in [56] vorgeschlagen wird. Umgekehrt bieten Landschafts- oder Stadtmetaphern die Möglichkeit für Nutzergruppen mit unterschiedlichem Detailwissen, eine gemeinsame Sprache zur Benennung von Systemstrukturen zu finden. Nützlich kann dies z. B. für die Kommunikation zwischen Management und Softwareentwicklern über die Planung der Weiterentwicklung eines Softwaresystems sein.

9.1.7 Exploration von Informationsräumen

Softwarevisualisierungs-Werkzeuge bilden Artefakte des Softwaresystems in einen Darstellungsraum ab. Um Nutzern diese Informationen zugänglich zu machen, muss das Werkzeug Mechanismen zur Exploration des Darstellungsraums zur Verfügung stellen. Nutzer sollten bei der Exploration neben Detailinformationen über einzelne Softwareartefakte auch Überblicksinformationen erhalten, damit sie jederzeit ihre Position im Darstellungsraum bestimmen können.

Ein Beispiel für ein sinnvolles Konzept für die Exploration von hierarchischen Informationsräumen ist in dem Softwarevisualisierungs-Werkzeug SHriMP/Creole [97] implementiert. Die hierarchische Struktur von Java-Programmen (Paket, Klasse, Methode/Attribut) wird im 2 Dimensionalen als

verschachtelter Graph dargestellt. Knoten werden durch geometrische Symbole repräsentiert und können ihre Kindknoten verkleinert innerhalb des Symbols anzeigen oder diese verdecken. Hierarchieebenen und damit Abstraktionsebenen lassen sich somit ein- und ausblenden.

SHriMP/Creole stellt Navigationstechniken zur Verfügung, welche durch Wahrung eines Überblickverständnisses die Nutzerorientierung unterstützen. Nutzer können durch Selektieren eines Knotens oder einer Kante, die zu diesem Knoten hinführt, zu dem Zielknoten navigieren und dessen Details betrachten. Die Navigation ist als Animation vom Start- bis zum Zielknoten realisiert. Entlang der kürzesten Kantenfolge im Graph vom Start- bis zum Zielknoten wird schrittweise von Knoten zu Knoten animiert. Da der Graph einen hierarchischen Informationsraum repräsentiert, finden Übergänge zwischen Knoten nur zwischen direkt aufeinander folgenden Hierarchieebenen oder auf der gleichen Ebene statt. Diese Technik ermöglicht es, von einer detaillierten Ansicht auf ein visualisiertes Element aus (z. B. einer Methode einer Java-Klasse) *aufzutauchen*, um über eine Übersicht (z. B. der Paketstruktur) wieder in eine detaillierte Ansicht (z. B. einer Methode einer Klasse aus einem anderen Paket) *einzutauchen*. Auf diese Weise lassen sich Aufrufbeziehungen von Klassen nachvollziehen, ohne dass die Nutzerorientierung verloren geht.

Die beschriebene Technik ist sehr gut zur Darstellung und Exploration von großen hierarchischen Informationsräumen geeignet. Bei der praktischen Anwendung des Werkzeugs SHriMP/Creole werden jedoch Performanz-Probleme offensichtlich. Die Umsetzung der Darstellungs- und Explorationstechnik in SHriMP/Creole müsste daher für die Untersuchung komplexer Softwaresysteme optimiert werden.

9.2 Zusammenfassung

Softwarevisualisierung bietet vielfältige Möglichkeiten, den Prozess der Verständnisgewinnung über ein komplexes Softwaresystem zu unterstützen. Trotzdem hat sich bisher kein implementierungsbasiertes Softwarevisualisierungs-Werkzeug als ein Standard durchsetzen können. Schwierigkeiten bei der Entwicklung eines Werkzeugs zur Visualisierung von komplexen Softwaresystemen bestehen insbesondere in der Umsetzung der folgenden Aspekte:

- Skalierbarkeit
- Automatisierbarkeit
- Explorationsmechanismen

Bei der Entwicklung eines Softwarevisualisierungs-Werkzeugs sollten diese drei Aspekte eine besondere Berücksichtigung finden. Zudem sollte die verwendete Visualisierungstechnik optimal an die jeweilige zu beantwortende softwaretechnische Fragestellung angepasst sein. Für Aufgaben wie die Fehlersuche oder die Prüfung von Quellcodeüberdeckung, bei denen der Quellcode primärer Untersuchungsgegenstand ist und weniger ein Strukturverständnis gefordert ist, sind quellcodebasierte Visualisierungstechniken vorzuziehen. Für die Darstellung der Systemstruktur ohne ausgeprägte Hierarchie bieten sich kräftebasierte Visualisierungstechniken an, bei denen sich die Struktur auf Basis von Softwareproduktmetriken ausbildet. Für eine hierarchische Darstellung bieten sich zum einen Techniken an, wie sie in SHriMP [97] verwendet werden. Zum anderen können Landschafts- und Stadtmetaphern bei der Visualisierung hierarchischer Strukturen sinnvoll sein. Eine gewichtige Rolle für die Wahl der Repräsentation der darzustellenden Informationen spielt neben der Skalierbarkeit die Frage, inwiefern die Semantik der Repräsentationen für eine intuitive Navigation im Darstellungsraum genutzt werden kann.

Zukünftige Forschungsaktivitäten im Bereich Softwarevisualisierung können insbesondere dann sinnvoll sein, wenn sie Fragestellungen adressieren, die nicht über die direkte Quellcodebetrachtung beantwortet werden können. Dies ist zum einen der Fall, wenn für die Verständnisgewinnung auf große Informationsmengen zurück gegriffen werden muss. Zum anderen kann Softwarevisualisierung sinnvoll sein, wenn für die Beantwortung der Fragestellungen der Quellcode aus Datenquelle nicht ausreicht und zusätzlich Laufzeitinformationen oder Informationen über weitere mit der Softwareentwicklung in Verbindung stehende Aspekte hinzugezogen werden müssen. Für weitere Forschung im Bereich Softwarevisualisierung lassen sich drei interessante Bereiche identifizieren. Diese sind a) die metrikbasierte Visualisierung statischer Aspekte großer Softwaresysteme, b) die Visualisierung dynamischer Aspekte von Softwaresystemen, die auf ein Verständnis der Systemstruktur zielt und c) die Visualisierung von Prozess-Management-Informationen, verknüpft mit Evolutionsinformationen über die Implementierung des Systems zur Optimierung des Entwicklungsprozesses.

Literaturverzeichnis

Literaturverzeichnis

[1] Agiles Manifest, http://agilemanifesto.org/.

[2] AspectBrowser, http://www-cse.ucsd.edu/users/wgg/Software/AB/.

[3] Aspect Mining Tool http://www.cs.ubc.ca/~jan/amt/.

[4] Augur, http://www.isr.uci.edu/projects/augur/.

[5] Ball, T., Eick, S., Software Visualization in the Large, IEEE Computer, 1996.

[6] Balzer, M., Noack, A., Deussen, O., Lewerentz, C., Software Landscapes: Visualizing Structure of Large Software Systems, Proceedings of Symposium on Visualization (VisSym), 2004.

[7] Bassil, S., Keller, R. K., Software Visualization Tools: Survey and Analysis, Proceedings of the 9th International Workshop on Program Comprehension (IWPC), 2001.

[8] Best, C., Michaud, J., Storey, M.A., „SHriMP: An Interactive Environment for Exploring Java Programs“, IWPC 2001 Workshop on Program Comprehension, 2001.

[9] Borland Together http://www.borland.com/together/.

[10] Boyack, K. W., Wylie, B. N., Davidson, G. S., Domain Visualization Using VxInsight for Science and Technology Management, Journal of the American Society for Information Science and Technology, 53. Jg. Heft 9, 2002.

[11] Chalmers, M., Using a Landscape Metaphor to Represent a Corpus of Documents, Proceedings of European Conference on Spatial Information Theory, 1993.

[12] Charters, St. M., Knight, C., Thomas, N., Munro, M., Visualisation for Informed Decision Making; From Code to Components, Proceedings of the Workshop on Software Engineering Decision Support (SEDECS), 2002.

[13] Chisel Group Members http://www.thechiselgroup.org/chisel_members/.

[14] Cleanscape Testwise, http://www.cleanscape.net/products/testwise/.

[15] CodeCrawler, http://www.iam.unibe.ch/~scg/Research/CodeCrawler/.

[16] CodePro AnalytiX http://www.instantiations.com/codepro/analytix/.

[17] Collberg, C., Kobourov, S., A System for Graph-Based Visualization of the Evolution of Software, Proceedings of the 2003 ACM symposium on Software visualization, 2003.

[18] CompuwareCorperation DevPartner, http://www.compuware.com/products/devpartner/.

[19] CrocoCosmos, BTU Cottbus, http://www-sst.informatik.tu-cottbus.de/CrocoCosmos/.

[20] Demeyer, S., Tichelaar, S., Steyaert, P., Definition of a common exchange model, http://www.iam.unibe.ch/~scg/Archive/famoos/FAMIX/InfExch11/INFXCH11.html, 1998.

[21] Demeyer, S., Ducasse, S., Tichelaar, S., Why unified is not universal. UML shortcomings for coping with round-trip engineering, Proceedings of the Second International Conference on The Unified Modeling Language (UML'99), 1999.

[22] Dourish, P., Froehlich, J., Augur – Supporting Software Development Visually, ISR Research Flier, 2003.

[23] Dourish, P., Froehlich, J., Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams, Proceedings of the 26th International Conference on Software Engineering, 2004.

[24] Ducasse, S., Lanza, M., Tichelaar, S., Moose: An extensible language-independent environment for reengineering object-oriented systems, Proceedings of the Second International Symposium on Constructing Software Engineering Tools, CoSET 2000.

[25] Ducasse, S., Lanza, M., Bertulli, R., High-Level Polymetric Views of Condensed Runtime Information, Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR), 2004.

[26] Dutton, R. T., Foster, J. C., Jack, M. A., Please Mind the Doors – Do interface meta-

phors improve the usability of voice response services? BT Technology Journal, 17. Jg. Heft 1, 1999.

[27] Eclipse Plattform, http://www.eclipse.org/.

[28] Eick, S., Steffen, J., Summner, E., SeeSoft – A Tool For Visualizing Line Oriented Software Statistics, IEEE Transactions on Software Engineering 1992.

[29] Eick, S., Nelson, C., Schmidt, D., Graphical Analysis of Computer Log Files, Communication of the ACM, 1994.

[30] Eick, S., A Visualization Tool for Y2K, IEEE Computer, 1998.

[31] Eick, S. G., Graves, T. L., Visualizing Software Changes, IEEE Transactions on Software Engineering 28(4), 2002.

[32] eXtreme Programming, http://www.extremeprogramming.org/.

[33] FAMOOS Projekt, The famoos objectoriented reengineering handbook, http://www.iam.unibe.ch/~scg/Archive/famoos/handbook/, 1999.

[34] Froehlich, J. E., Dourish, P., Unifying Artifacts and activities in a visual tool for distributed software teams, 26th International Conference on Software Engineering ICSE'04.

[35] FrontEndArt, http://www.frontendart.com/.

[36] Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software, Adisson-Wesley, 1995.

[37] Garcia, V. C., Lucredio, D., Do Prado, A. F., Towards an effective approach for Reverse Engineering, WCRE 2003.

[38] GNU General Public License http://www.gnu.org/copyleft/gpl.html.

[39] Griswold, W. G., Kato, Y., Yuan, J. J., AspectBrowser: Tool Support for Managing Dispersed Aspects, Department of Computer Science and Engineering, University of California, San Diego, Technical Report CS99-0640, 1999.

[40] Headway reView, http://www.headwaysoft.com/.

[41] Hannemann, J., Kicyales, G., Overcoming the Prevalent Decomposition in Legacy Code, Proceedings of Advanced Separation of Concerns Workshop, International Conference on Software Engineering, 2001.

[42] Harrold, M. J., Jones, J. A., Stasko, J. T., Visualization for Fault Localization, Proceedings of the ICSE 2001 Workshop on Software Visualization, 2001.

[43] Harrold, M. J., Jones, J. A., Stasko, J. T., Visualization of Test Information to Assist Fault Localization, Proceedings of the 24th International Conference on Software Engineering, 2002.

[44] Harrold, M. J., Jones, J. A., Orso, A., Visualization of Program-Execution Data for Deployed Software, Proceedings of the ACM Symposium on Software Visualization, 2003.

[45] IBM Rational Software http://www-306.ibm.com/software/rational/.

[46] IBM Rational Purify http://www-306.ibm.com/software/awdtools/purify/.

[47] Ingram, R., Benford, St., Legibility Enhancement for Information Visualisation, Proceedings of the 6th conference on Visualization, 1995.

[48] Irwin, W., Churcher, N., Object Oriented Metrics: Precision Tools and Configurable Visualisations, Ninth International Software Metrics Symposium (METRICS'03), 2003.

[49] jEdit, http://www.jedit.org/.

[50] jEdit Features, http://www.jedit.org/index.php?page=features.

[51] jEdit Plugins, http://plugins.jedit.org/.

[52] jEdit User's Guide, http://www.jedit.org/42docs/users-guide/index.html.

[53] jEdit 4.2 API Documentation, http://www.jedit.org/42docs/api/index.html.

[54] Jones, J., Orso, A., Harrold, M. J., Gammatella: Visualizing Program-Execution Data for Deployed Software, Information Visualization 2004, Volume 3, Number 3, pages 173-188, Palgrave Macmillan Journals, 2004.

[55] Jünger, M., Mutzel, P., Automatisches Layout von Diagrammen, 2001. http://aragorn.ads.tuwien.ac.at/people/Mutzel/publications/ORNEWS/ORNEWS.htm.

[56] Knight, C., Virtual Software in Reality, Ph.D. Thesis, Durham (UK) 2000.

[57] Knight, C., Munro, M., Mindless Visualisations, Proceedings of the 6th ERCIM "User Interfaces for All" Workshop, 2000.

[58] Koschke, R., Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey, Journal on Software Maintenance and Evolution, John Wiley & Sons, Ltd., Vol. 15, No. 2, S. 87-109, 2003.

[59] Krusch, J., Reckziegel, J., Albrecht, M., Agiles Projektmanagement in einem MDA-basierten Projekt: Erfahrungsbericht, OBJEKTspektrum 01/2005.

[60] Lanza, M., Combining Metrics and Graphs for Object Oriented Reverse Engineering, Master Thesis, Universität Bern, 1999.

[61] Lanza, M., The Evolution Matrix: Recovering Software Evolution using Software Visualization Techniques, Proceedings of the 4th International Workshop on Principles of Software Evolution, 2001.

[62] Lanza, M., Object-Oriented Reverse Engineering, Coarse-grained, Fine-grained, and Evolutionary Software Visualization, Ph.D. Thesis, Universität Bern, 2003.

[63] LFTP, http://lftp.yar.ru/.

[64] Lintern, R., Michaud, J., Storey, M.-A.D., Wu, X., Plugging-in Visualization: Experiences Integrating a Visualization Tool with Eclipse, Proceedings of the 2003 ACM symposium on Software visualization, 2003.

[65] Lintern, R., Murray, A., Storey, M.-A.D., Wu, X., Designing and Evaluating an Interactive Visualization Tool to Support Version Control of Software, Vissoft 2003.

[66] Lorenz, M., Kidd, J., Object-oriented software metrics: a practical guide, Prentice-Hall Object-Oriented Series, 1994.

[67] Lynch, K., The Image of the City. The MIT Press, 1960.

[68] MacKinlay, J. D., Automating the design of graphical presentation of relational information. In: ACM Transaction on Graphics, 5. Jg. Heft 2, 1986.

[69] Maletic, J. I., Marcus, A., Feng, L., Source Viewer 3D (sv3D) – A Framework for Software Visualization, Proceedings of the 25th IEEE/ACM International Conference on Software Engineering, 2003.

[70] Maletic, J .I., Marcus, A., Feng, L., Comprehension of Software Analysis Data Using 3D Visualization, Proceedings of the IEEE International Workshop on Program Comprehension, 2003.

[71] Maletic, J. I., Marcus, A., Feng, L., 3D Representations for Software Visualization, Proceedings of the ACM Symposium on Software Visualization, 2003.

[72] [MarcusFengMaletic2003] Marcus, Feng, Maletic: 3D Representations for Software Visualization. In: Proceedings of the ACM Symposium on Software Visualization (SoftVis), 2003.

[73] Mawl – A Language for Web Service Programming, http://www.bell-labs.com/project/MAWL/.

[74] Michaud, J., Storey, M.-A.D., Müller, A., Integrating Information Sources for Visualizing Java Programs, Proceedings of the International Conference of Software Maintenance (ICSM'2002), 2002.

[75] Object Management Group, Introduction To OMG's Unified Modelling Language, http://www.omg.org/gettingstarted/what_is_uml.htm.

[76] Object Management Group, OMG Model Driven Architecture, http://www.omg.org/mda/.

[77] OpenCMS, http://www.opencms.org/.

[78] Panas, Th., Berrigan, R., Grundy, J., A 3D Metaphor for Software Production Visualization. In: Proceedings of Seventh International Conference on Information Visualization (IV 2003), 2003.

[79] Piccolo Zooming-Bibliothek http://www.cs.umd.edu/hcil/piccolo/.

[80] Ploix, D., Analogical Representations of Programs, Proceedings of First International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT), 2002.

[81] Price, B., Baecker, R., Small, I., A Principled Taxonomy of Software Visualization, In Journal of Visual Languages and Computing, Vol. 4, No. 3, S. 211-266, 1993.

[82] Protégé, http://protege.standford.edu/.

[83] Reiss, S. P., Bee/Hive: A Software Visualization Back End, IEEE Workshop on Software Visualization, 2001.

[84] Reiss, S. P., Software Visualization 2005, Call for Papers, http://www.softvis.org/softvis05/.

[85] Rekimoto, J., Green, M., The Information Cube, Using Transparency in 3D Information Visualization. In: Proceedings of the Third Ann. Workshop Information Technologies & Systems (WITS '93), 1993.

[86] Rigi, http://www.rigi.csc.uvic.ca/.

[87] Santos, C., Gros, P., Abel, P., Paris, J. P., Mapping Information onto 3D Virtual Worlds, Proceedings of the IEEE International Conference on Information Visualization (IV'00), 2000.

[88] SCG Uni Bern, http://www.iam.unibe.ch/.

[89] Schäftlein, R., Mahler, T., MDA in der Praxis, Erfahrungen mit modellgetriebener Softwareentwicklung in Großprojekten bei

der Karstadt Warenhaus AG, Proceedings of Software Engineering (SE05) 2005.

[90] Schnadwinkel, B., Neue Medien – neue Metaphern? Sprachliche Erschließung des neuen Mediums Internet durch Metaphern (deutsch-französisch), Magisterarbeit, Hamburg 2002.

[91] Schumann, H., Müller, W., Visualisierung – Grundlagen und allgemeine Methoden, Springer-Verlag, Heidelberg 2000.

[92] SeeSoft http://www.cc.gatech.edu/computing/classes/cs7390_98_winter/reports/realsys/seesoft.html.

[93] Shneiderman, B., Tree Visualization with Tree-maps: A 2-d space-filling approach. *ACM Trans. Graphics,* vol. 11, no. 1, pp. 92-99, 1992.

[94] SHriMP, http://www.chiselgroup.com/SHriMP/.

[95] Sommerville, I., Software Engineering, fourth ed. Addison Wesley, 1992.

[96] Stasko, J. T., et al., Software Visualization – Programming as a Multimedia Experience, The MIT Press, 1998.

[97] Storey, M.-A.D., A Cognitive Framework for Describing and Evaluation Software Exploration Tools, Ph.D. Thesis, 1998.

[98] Storey, M.-A.D., Wong, K., Müller, H.A.., How Do Program Understanding Tools Affect How Programmers Understand Programs? Proceedings of the Fourth Working Conference on Reverse Engineering, p. 12-21, 1999.

[99] Storey, M.A., "SHriMP Views: An Interactive Environment for Exploring Multiple Hierarchical Views of a Java Program.", Workshop on Software Visualization (ICSE), 2001.

[100] Storey, M.A. et al., „SHriMP Views: An Interactive Environment for Information Visualization and Navigation", Proceedings of CHI, 2002.

[101] Taylor, C. M. B., Munro, M., Revision Towers, Vissoft 2002.

[102] The Java Tutorial, The Reflection API, http://java.sun.com/docs/books/tutorial/reflect/.

[103] Telea, A., Maccari, A., Riva, C., An Open Toolkit for Prototyping Reverse Engineering Visualizations, Proceedings of the symposium on Data Visualisation, 2002.

[104] Tesler, J. D., Strasnick, S. L., FSN – The 3D file system navigator. Silicon Graphics, Inc., 1992. http://www.sgi.com/fun/freeware/3d_navigator.html.

[105] UML, http://www.uml.org/.

[106] Virtual Rendering System, http://www.vrs3d.org/.

[107] Wu, X., Storey, M.-A., Visualization to Support Version Control Software: Suggested Requirements, Vissoft 2003.

[108] Wyatt, J. B., Software Visualization and Program Understanding, Department of Information Sciences, University of Pittsburgh, 1999.

[109] Young, P., Munro, M., Visualizing Software in Virtual Reality, Proceedings of the 6th International Workshop on Program Comprehension (IWPC '98), 1998.

[110] Zehler, T., Software-Visualisierung, Report: ViSEK/045/D, 2004.

KONZEPTE DER SOFTWAREVISUALISIERUNG FÜR KOMPLEXE, OBJEKTORIENTIERTE SOFTWARESYSTEME

Autorenverzeichnis

Autorenverzeichnis

Bohnet, Johannes	bohnet@hpi.uni-potsdam.de
Brinkmann, Daniel	daniel.brinkmann@hpi.uni-potsdam.de
Döllner, Jürgen	doellner@hpi.uni-potsdam.de
Gierak, Alexander	alexander.gierak@hpi.uni-potsdam.de
Hagedorn, Benjamin	benjamin.hagedorn@hpi.uni-potsdam.de
Lazic, Nebojsa	lazic.nebojsa@arcor.de
Schöbel, Michael	michael.schoebel@hpi.uni-potsdam.de

Universität Potsdam
Hasso-Plattner-Institut für Softwaresystemtechnik GmbH
Fachgebiet Computergrafische Systeme

Adresse: Prof.-Dr.-Helmert-Str. 2-3
14482 Potsdam
Telefon: +49 (0) 331 5509-170
Fax: +49 (0) 331 5509-179

ISBN 3-937786-54-6
ISSN 1613-5652